

实战 Python 网络爬虫

从爬虫软件开发到自己动手开发爬虫框架

黄永祥 / 著

从原理到实践，深入浅出，热门爬虫核心技术全掌握
涵盖丰富的爬虫工具、库、框架，十余个实战项目
资深爬虫工程师倾力奉献，入门、进阶、求职必备

清华大学出版社



清华大学出版社
北 京

内 容 简 介

本书从原理到实践，循序渐进地讲述了使用 Python 开发网络爬虫的核心技术。全书从逻辑上可分为基础篇、实战篇和爬虫框架篇三部分。基础篇主要介绍了编写网络爬虫所需的基础知识，包括网站分析、数据抓取、数据清洗和数据入库。网站分析讲述如何使用 Chrome 和 Fiddler 抓包工具对网站做全面分析；数据抓取介绍了 Python 爬虫模块 Urllib 和 Requests 的基础知识；数据清洗主要介绍字符串操作、正则和 BeautifulSoup 的使用；数据入库讲述了 MySQL 和 MongoDB 的操作，通过 ORM 框架 SQLAlchemy 实现数据持久化，进行企业级开发。实战篇深入讲解了分布式爬虫、爬虫软件的开发、12306 抢票程序和微博爬取等。框架篇主要讲述流行的爬虫框架 Scrapy，并以 Scrapy 与 Selenium、Splash、Redis 结合的项目案例，让读者深层次了解 Scrapy 的使用。此外，本书还介绍了爬虫的上线部署、如何自己动手开发一款爬虫框架、反爬虫技术的解决方案等内容。

本书使用 Python 3.X 编写，技术先进，项目丰富，适合欲从事爬虫工程师和数据分析师岗位的初学者、大学生和研究生使用，也很适合有一些网络爬虫编写经验，但希望更加全面、深入理解 Python 爬虫的开发人员使用。

图书在版编目（CIP）数据

实战 Python 网络爬虫 / 黄永祥著.—北京：清华大学出版社，2019
ISBN 978-7-302-52489-2

Ⅰ. ①实… Ⅱ. ①黄… Ⅲ. ①软件工具－程序设计 Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字（2019）第 043080 号

责任编辑： 王金柱
封面设计： 王　翔
责任校对： 闫秀华
责任印制： 沈　露

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦 A 座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
印 装 者： 清华大学印刷厂
经　　销： 全国新华书店
开　　本： 190mm×260mm　　**印　　张：** 30.25　　**字　　数：** 774 千字
版　　次： 2019 年 6 月第 1 版　　**印　　次：** 2019 年 6 月第 1 次印刷
定　　价： 99.00 元

产品编号：082567-01

前　言

随着大数据和人工智能的普及，Python 的地位也变得水涨船高，许多技术人员投身于 Python 开发，其中网络爬虫是 Python 最为热门的应用领域之一。在爬虫领域，Python 可以说是处于霸主地位，Python 能解决爬虫开发过程中所遇到的难题，开发速度快且支持异步编程，大大缩短了开发周期。此外，从事数据分析的工程师，为获取数据，很多时候也会用到网络爬虫的相关技术，因此，Python 爬虫编程已成为爬虫工程师和数据分析师的必备技能。

本书结构

本书共分 28 章，各章内容概述如下：

第 1 章介绍什么是网络爬虫、爬虫的类型和原理、爬虫搜索策略和爬虫的合法性及开发流程。

第 2 章讲解爬虫开发的基础知识，包括 HTTP 协议、请求头和 Cookies 的作用、HTML 的布局结构、JavaScript 的介绍、JSON 的数据格式和 Ajax 的原理。

第 3 章介绍使用 Chrome 开发工具分析爬取网站，重点介绍开发工具的 Elements 和 Network 标签的功能和使用方式，并通过开发工具分析 QQ 网站。

第 4 章主要介绍 Fiddler 抓包工具的原理和安装配置，Fiddler 用户界面的各个功能及使用方法。

第 5 章讲述了 Urllib 在 Python 2 和 Python 3 的变化及使用，包括发送请求、使用代理 IP、Cookies 的读写、HTTP 证书验收和数据处理。

第 6 章~第 8 章介绍 Python 第三方库 Requests、Requests-Cache 爬虫缓存和 Requests-HTML，包括发送请求、使用代理 IP、Cookies 的读写、HTTP 证书验收和文件下载与上传、复杂的请求方式、缓存的存储机制、数据清洗以及 Ajax 动态数据爬取等内容。

第 9 章介绍网页操控和数据爬取，重点讲解 Selenium 的安装与使用，并通过实战项目“百度自动答题”，讲解了 Selenium 的使用。

第 10 章介绍手机 App 数据爬取，包括 Appium 的原理与开发环境搭建、连接 Android 系统，并通过实战项目“淘宝商品采集”，介绍了 App 数据的爬取技巧。

第 11 章介绍 Splash、Mitmproxy 与 Aiohttp 的安装和使用，包括 Splash 动态数据抓取、Mitmproxy 抓包和 Aiohttp 高并发抓取。

第 12 章介绍验证码的种类和识别方法，包括 OCR 的安装和使用、验证码图片处理和使用第三方平台识别验证码。

第 13 章讲述数据清洗的三种方法，包括字符串操作（截取、查找、分割和替换）、正则表达式的使用和第三方库 BeautifulSoup 的安装以及使用。

第 14 章讲述如何将数据存储到文件，包括 CSV、Excel 和 Word 文件的读取和写入方法。

第 15 章介绍 ORM 框架 SQLAlchemy 的安装及使用，实现关系型数据库持久化存储数据。

第 16 章讲述非关系型数据库 MongoDB 的操作，包括 MongoDB 的安装、原理和 Python 实现

MongoDB 的读写。

第 17 章至第 21 章介绍了 5 个实战项目，分别是：爬取 51Job 招聘信息、分布式爬虫——QQ 音乐、12306 抢票爬虫、微博爬取和微博爬虫软件的开发。

第 22 章至第 25 章介绍了 Scrapy 爬虫框架，包括 Scrapy 的运行机制、项目创建、各个组件的编写（Setting、Items、Item Pipelines 和 Spider）和文件下载及 Scrapy 中间件，并通过实战项目"Scrapy+Selenium 爬取豆瓣电影评论"、"Scrapy+Splash 爬取 B 站动漫信息"和"Scrapy+Redis 分布式爬取猫眼排行榜"、"爬取链家楼盘信息"和"QQ 音乐全站爬取"，深入讲解了 Scrapy 的应用和分布式爬虫的编写技巧。

第 26 章介绍爬虫的上线部署，包括非框架式爬虫和框架式爬虫的部署技巧。

第 27 章介绍常见的反爬虫技术，并给出了可行的反爬虫解决方案。

第 28 章介绍爬虫框架的编写，学习如何自己动手编写一款爬虫框架，以满足特定业务场景的需求。

本书特色

循序渐进，涉及面广：本书站在初学者的角度，循序渐进地介绍了使用 Python 开发网络爬虫的各种知识，内容由浅入深，几乎涵盖了目前网络爬虫开发的各种热门工具和前瞻性技术。

实战项目丰富，扩展性强：本书采用大量的实战项目进行讲解，力求通过实际应用使读者更容易地掌握爬虫开发技术，以应对业务需求。本书项目经过编者精心设计和挑选，根据实际开发经验总结而来，涵盖了在实际开发中所遇到的各种问题。对于精选项目，尽可能做到步骤详尽、结构清晰、分析深入浅出，而且案例的扩展性强，读者可根据实际需求扩展开发。

从理论到实践，注重培养爬虫开发思维：在讲解过程中，不仅介绍理论知识，注重培养读者的爬虫开发思维，而且安排了综合应用实例或小型应用程序，使读者能顺利地将理论应用到实践中。

特色干货，倾情分享：本书大部分内容都来自作者多年来的编程实践，操作性很强。值得关注的是，本书还介绍了爬虫软件和爬虫框架的开发，供学有余力的读者扩展知识结构，提升开发技能。

源代码下载

本书所有程序代码均在 Python 3.6 下调试通过，源代码 Github 下载地址：

https://github.com/xyjw/python-Reptile

你也可以扫描下面的二维码下载。



如果你在下载过程中遇到问题，可发送邮件至 554301449@qq.com 获得帮助，邮件标题为"实

战 Python 网络爬虫下载资源”。

技术服务

读者在学习或者工作的过程中，如果遇到实际问题，可以加入 QQ 群 93314951 与笔者联系，笔者会在第一时间给予回复。

读者对象

本书主要适合以下读者阅读：

- Python 网络爬虫初学者及在校学生。
- Python 初级爬虫工程师。
- 从事数据抓取和分析的技术人员。
- 学习 Python 程序设计的开发人员。

虽然笔者力求本书更臻完美，但由于水平所限，难免会出现错误，特别是实例中爬取的网站可能随时更新，导致源码在运行过程中出现问题，欢迎广大读者和高手专家给予指正，笔者将十分感谢。

黄永祥
2019 年 1 月

目　录

第 1 章

理解网络爬虫

1.1 爬虫的定义

网络爬虫是一种按照一定的规则自动地抓取网络信息的程序或者脚本。简单来说，网络爬虫就是根据一定的算法实现编程开发，主要通过 URL 实现数据的抓取和发掘。

随着大数据时代的发展，数据规模越来越庞大，数据类型繁多，但是数据价值普遍较低。为了从庞大的数据体系里获取有价值的数据，从而延伸了网络爬虫、数据分析等多个职位。近几年，网络爬虫的需求更是井喷式地爆发，在招聘的供求市场上往往是供不应求，造成这个现状的主要原因就是求职者的专业水平低于需求企业的要求。

传统的爬虫有百度、Google、必应等搜索引擎，这类通用的搜索引擎都有自己的核心算法。但是，通用的搜索引擎存在着一定的局限性：

（1）不同的搜索引擎对于同一个搜索会有不同的结果，搜索出来的结果未必是用户需要的信息。

（2）通用的搜索引擎扩大了网络覆盖率，但有限的搜索引擎服务器资源与无限的网络数据资源之间的矛盾将进一步加深。

（3）随着网络上数据形式繁多和网络技术的不断发展，图片、数据库、音频、视频多媒体等不同数据大量出现，通用搜索引擎往往对这些信息含量密集且具有一定结构的数据无能为力，不能很好地发现和获取。

因此，为了得到准确的数据，定向抓取相关网页资源的聚焦爬虫应运而生。聚焦爬虫是一个自动下载网页的程序，可根据设定的抓取目标有目的性地访问互联网上的网页与相关的 URL，从而获取所需要的信息。与通用爬虫不同，聚焦爬虫并不追求全面的覆盖率，而是抓取与某一特定内容相关的网页，为面向特定的用户提供准备数据资源。

1.2 爬虫的类型

网络爬虫根据系统结构和开发技术大致可以分为 4 种类型：通用网络爬虫、聚焦网络爬虫、增量式网络爬虫和深层网络爬虫。

通用网络爬虫又称全网爬虫，常见的有百度、Google、必应等搜索引擎，爬行对象从一些初始 URL 扩充到整个网站，主要为门户站点搜索引擎和大型网站服务采集数据，具有以下特点：

（1）由于商业原因，引擎的算法是不会对外公布的。

（2）这类网络爬虫的爬取范围和数量巨大，对于爬取速度和存储空间要求较高，爬取页面的顺序要求相对较低。

（3）待刷新的页面太多，通常采用并行工作方式，但需要较长时间才能刷新一次页面。

（4）存在一定缺陷，通用网络爬虫适用于为搜索引擎搜索广泛的需求。

聚焦网络爬虫又称主题网络爬虫，是选择性地爬取根据需求的主题相关页面的网络爬虫。与通用网络爬虫相比，聚焦爬虫只需要爬取与主题相关的页面，不需要广泛地覆盖无关的网页，很好地满足一些特定人群对特定领域信息的需求。

增量式网络爬虫是指对已下载网页采取增量式更新和只爬取新产生或者已经发生变化的网页的爬虫，它能够在一定程度上保证所爬取的页面尽可能是新的页面。只会在需要的时候爬取新产生或发生更新的页面 ，并不重新下载没有发生变化的页面，可有效减少数据下载量，及时更新已爬取的网页，减小时间和空间上的耗费，但是增加了爬取算法的复杂度和实现难度，基本上这类爬虫在实际开发中不太普及。

深层网络爬虫是大部分内容不能通过静态 URL 获取的、隐藏在搜索表单后的、只有用户提交一些关键词才能获得的网络页面。例如某些网站需要用户登录或者通过提交表单实现提交数据。这类爬虫也是本书讲述的重点之一。

实际上，聚焦网络爬虫、增量式网络爬虫和深层网络爬虫可以通俗地归纳为一类，因为这类爬虫都是定向爬取数据。相比于通用爬虫，这类爬虫比较有目的性，也就是网络上经常说的网络爬虫，而通用爬虫在网络上通常称为搜索引擎。

1.3 爬虫的原理

通用网络爬虫的实现原理及过程如图 1-1 所示。

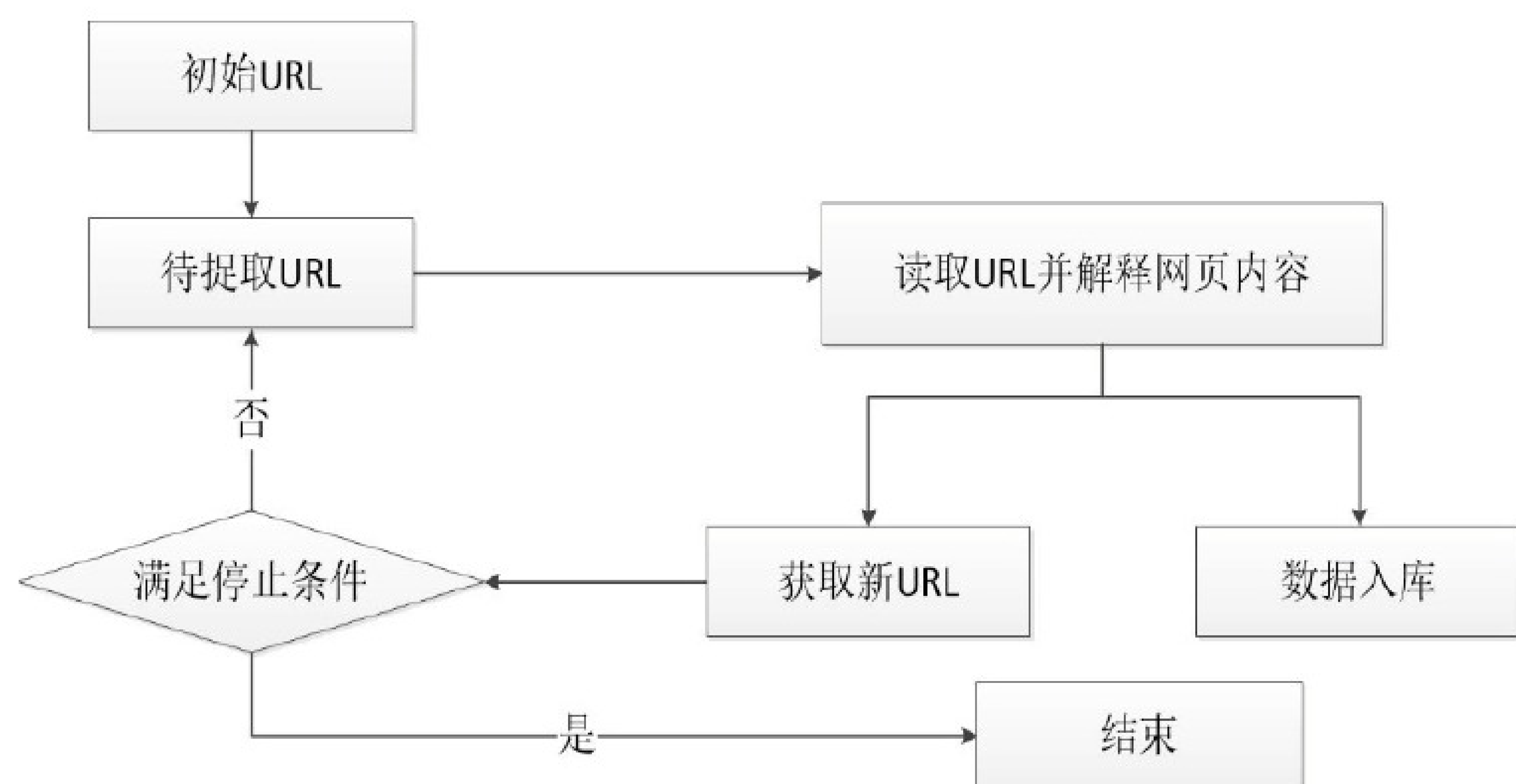


图 1-1　通用爬虫实现的原理及过程

通用网络爬虫的实现原理：

（1）获取初始的 URL。初始的 URL 地址可以人为地指定，也可以由用户指定的某个或某几个初始爬取网页决定。

（2）根据初始的 URL 爬取页面并获得新的 URL。获得初始的 URL 地址之后，先爬取当前 URL 地址中的网页信息，然后解析网页信息内容，将网页存储到原始数据库中，并且在当前获得的网页信息里发现新的 URL 地址，存放于一个 URL 队列里面。

（3）从 URL 队列中读取新的 URL，从而获得新的网页信息，同时在新网页中获取新 URL，并重复上述的爬取过程。

（4）满足爬虫系统设置的停止条件时，停止爬取。在编写爬虫的时候，一般会设置相应的停止条件，爬虫则会在停止条件满足时停止爬取。如果没有设置停止条件，爬虫就会一直爬取下去，一直到无法获取新的 URL 地址为止。

聚焦网络爬虫的执行原理和过程与通用爬虫大致相同，在通用爬虫的基础上增加两个步骤：定义爬取目标和筛选过滤 URL，原理如图 1-2 所示。

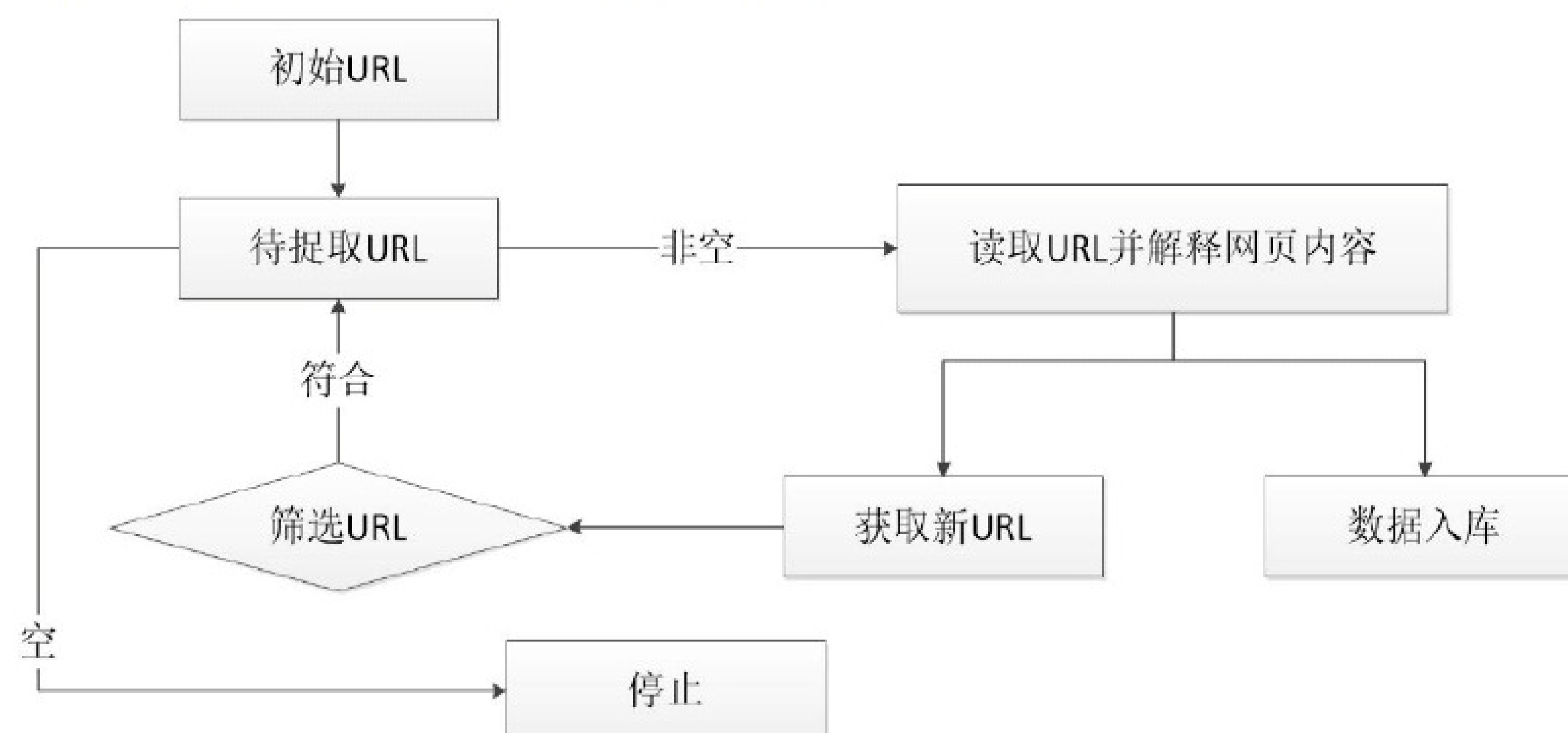


图 1-2　聚焦网络爬虫的原理

聚焦网络爬虫的实现原理：

（1）制定爬取方案。在聚焦网络爬虫中，首先要依据需求定义聚焦网络爬虫爬取的目标以及整体的爬取方案。

（2）设定初始的 URL。

（3）根据初始的 URL 抓取页面，并获得新的 URL。

（4）从新的 URL 中过滤掉与需求无关的 URL，将过滤后的 URL 放到 URL 队列中。

（5）在 URL 队列中，根据搜索算法确定 URL 的优先级，并确定下一步要爬取的 URL 地址。因为聚焦网络爬虫具有目的性，所以 URL 的爬取顺序不同会导致爬虫的执行效率不同。

（6）得到新的 URL，将新的 URL 重现上述爬取过程。

（7）满足系统中设置的停止条件或无法获取新的 URL 地址时，停止爬行。

1.4 爬虫的搜索策略

在互联网数据时代，有三大搜索策略需要有所了解，下面一一介绍。

1. 深度优先搜索

深度优先搜索是在开发爬虫早期使用较多的方法，目的是达到被搜索结构的叶结点（那些不包含任何超级 URL 的 HTML 文件）。在一个 HTML 文件中，当一个 URL 被选择后，被选 URL 将执行深度优先搜索，搜索后得到新的 HTML 文件，再从新的 HTML 获取新的 URL 进行搜索，以此类推，不断地爬取 HTML 中的 URL，直到 HTML 中没有 URL 为止。

深度优先搜索沿着 HTML 文件中的 URL 走到不能再深入为止，然后返回到某一个 HTML 文件，再继续选择该 HTML 文件中的其他 URL。当不再有其他 URL 可选择时，说明搜索已经结束。其优点是能遍历一个 Web 站点或深层嵌套的文档集合。缺点是因为 Web 结构相当深，有可能造成一旦进去再也出不来的情况发生。

举个例子，比如一个网站的首页里面带有很多 URL，深度优先通过首页的 URL 进入新的页面，然后通过这个页面里的 URL 再进入新的 URL，不断地循环下去，直到返回的页面没有 URL 为止。如果首页有两个 URL，选择第一个 URL 后，生成新的页面就不会返回首页，而是在新的页面选择一个新的 URL，这样不停地访问下去。

2. 宽度优先搜索

宽度优先搜索是搜索完一个 Web 页面中所有的 URL，然后继续搜索下一层，直到底层为止。例如，首页中有 3 个 URL，爬虫会选择其中之一，处理相应的页面之后，然后返回首页再爬取第二个 URL，处理相应的页面，最后返回首页爬取第三个 URL，处理第三个 URL 对应的页面。

一旦一层上的所有 URL 都被选择过，就可以开始在刚才处理过的页面中搜索其余的 URL，这就保证了对浅层的优先处理。当遇到一个无穷尽的深层分支时，不会导致陷进深层文档中出不来的情况发生。宽度优先搜索策略还有一个优点，能够在两个页面之间找到最短路径。

宽度优先搜索策略通常是实现爬虫的最佳策略，因为它容易实现，而且具备大多数期望的功能。但是如果要遍历一个指定的站点或者深层嵌套的 HTML 文件集，用宽度优先搜索策略就需要花费较长时间才能到达最底层。

3. 聚焦爬虫的爬行策略

聚焦爬虫的爬行策略只针对某个特定主题的页面，根据“最好优先原则”进行访问，快速、

有效地获得更多与主题相关的页面，主要通过内容与Web的URL结构指导进行页面的抓取。聚焦爬虫会给所下载的页面一个评价分，根据得分排序插入一个队列中。最好下一个搜索对弹出队列的第一个页面进行分析后执行，这种策略保证爬虫能优先跟踪那些最有可能URL到目标页面的页面。

决定网络爬虫搜索策略的关键是评价URL价值，即URL价值的计算方法，不同的价值评价方法计算出的URL的价值不同，表现出的URL的“重要程度”也不同，从而决定不同的搜索策略。由于URL包含于页面之中，而通常具有较高价值的页面包含的URL也具有较高价值，因此对URL价值的评价有时也转换为对页面价值的评价。

1.5　爬虫的合法性与开发流程

网络爬虫在大多数情况下都不会违法，在生活中几乎都有爬虫应用，比如在百度中搜索的内容几乎都是通过爬虫采集下来的，因此网络爬虫作为一门技术，技术本身是不违法的，且在大多数情况下可以放心使用爬虫技术。当然也有特殊情况，正如水果刀本身在法律上并不被禁止使用，但是用来伤害他人，这就触犯了法律规则。一般情况下，爬虫所带来的违法风险主要体现在以下几个方面：

（1）利用爬虫技术与黑客技术结合，攻击网站后台，从而窃取后台数据。因为爬虫是爬取网站上的网页信息，这些信息能给用户浏览，也就是说这些信息允许我们使用和爬取。但网站的后台数据是不被公开的数据，这些数据涉及了用户的隐私和财产安全，如果通过爬虫技术与黑客技术窃取后台数据，这就明显触发法律的底线。

（2）利用爬虫恶意攻击网站，造成网站系统的瘫痪。爬虫是通过程序去访问并操控网站，因此访问速度非常快，再加上程序的高并发处理，可以在短时间内模拟成千上万的用户在访问网站。当网站的访问量过高，就会加重网站的负载，从而造成系统的瘫痪，如果长期这样恶意攻击网站系统，也很可能违反相关的法律条例。

综上所述，爬虫技术本身是无罪的，问题往往出在人的无限欲望上。因此爬虫开发者和企业经营者的道德良知才是避免触碰法律底线的根本所在。

既然爬虫技术是合法的，那么，我们有必要了解一下爬虫的开发流程。只有掌握开发流程，才能编写高质的爬虫程序，这好比盖房子一样，建筑施工人员需要根据房屋设计图才能搭建房子，而房屋设计图等同于爬虫的开发流程。一般情况下，爬虫的开发流程如下：

（1）需求说明。任何程序开发都离不开需求说明，爬虫开发也是如此。需求说明包含功能说明、功能的业务逻辑等详细说明。爬虫的需求说明要明确告知开发人员需要爬取哪些数据、数据的存储方式以及爬虫的爬取效率。

（2）爬虫开发计划。根据爬虫的需求说明制定相关的开发计划，比如选择爬虫的开发工具、功能模块化设计、设计爬虫运行模式等一系列开发明细。

（3）爬虫的功能开发。根据开发计划编写相应的功能代码。以功能模块化设计为依据，每个功能模块以函数或类的形式表示，再将各个模块进行组合，从而实现整个爬虫功能的开发。

（4）爬虫的部署与交付。程序开发完成后（包含测试通过）就可以进行部署上线或交付客户。

部署和交付的方式有多种，比如打包 exe 程序、GUI 界面（爬虫软件）或定时执行等。

上述的爬虫开发流程是相对而言的，每一个开发步骤并非一成不变的，具体的开发流程还需要结合实际情况而定。

1.6 本章小结

网络爬虫的类型理论上分为 4 类，但实际上主要是两大类：通用爬虫和聚焦爬虫。通用爬虫主要有 Google、百度、必应等搜索引擎，主要以核心算法为主导，学习成本相对较高。聚焦爬虫就是定向爬取数据，是有目的性的爬虫，学习成本相对较低。

我们常说的网络爬虫大多数以聚焦爬虫为主，其原理和过程与通用爬虫大致相同，读者在编写爬虫程序的时候，需要以设定的爬虫规则和爬取目标为主导，这样更具较强的目的性。

网络爬虫在大多数情况下都不会违法，在生活中几乎都有爬虫应用，比如在百度中搜索的内容几乎都是通过爬虫采集下来的，因此网络爬虫作为一门技术，技术本身是不违法的，且在大多数情况下可以放心使用爬虫技术。当然也有特殊情况，正如水果刀本身在法律上并不被禁止使用，但是用来伤害他人，这就触犯了法律规则。

既然爬虫技术是合法的，那么，我们有必要了解爬虫的开发流程。只有掌握开发流程，才能编写高质的爬虫程序，这好比盖房子一样，建筑施工人员需要根据房屋设计图才能搭建房子，而房屋设计图等同于爬虫的开发流程。

第 2 章

爬虫开发基础

2.1 HTTP 与 HTTPS

HTTP（Hyper Text Transfer Protocol，超文本传输协议）是一个客户端和服务器端请求和应答的标准（TCP）。客户端是终端用户，服务器端是网站。通过使用 Web 浏览器、网络爬虫或者其他工具，客户端发起一个到服务器上指定端口（默认端口为 80）的 HTTP 请求，这个客户端叫用户代理（User Agent）。响应的服务器上存储着资源，比如 HTML 文件和图像，这个服务器为源服务器（Origin Server），在用户代理和服务器中间可能存在多个中间层，比如代理、网关或者隧道（Tunnels）。

通常，由 HTTP 客户端发起一个请求，建立一个到服务器指定端口（默认是 80 端口）的 TCP 连接，HTTP 服务器则在那个端口监听客户端发送过来的请求，一旦收到请求，服务器（向客户端）发回一个状态行（比如"HTTP/1.1 200 OK"）和（响应的）消息，消息的消息体可能是请求的文件、错误消息或者其他一些信息。

在浏览器的地址栏输入的网站地址叫作 URL（Uniform Resource Locator，统一资源定位符）。就像每家每户都有一个门牌地址一样，每个网页也都有一个 Internet 地址。在浏览器的地址框中输入一个 URL 或单击一个超级 URL 时，URL 就确定了要浏览的地址，向服务器发送一次请求，浏览器通过超文本传输协议（HTTP）传送到服务器，服务器根据请求头做出相应的响应，将响应数据返回到客户端，客户端收到响应内容后，通过浏览器翻译成网页。

HTTP 协议传输的数据都是未加密的，也就是明文的数据，因此使用 HTTP 协议传输隐私信息非常不安全。为了保证这些隐私数据能加密传输，于是网景公司设计了 SSL（Secure Sockets Layer）协议用于对 HTTP 协议传输的数据进行加密，从而诞生了 HTTPS。

HTTPS（Hyper Text Transfer Protocol orer Secure Sokcket Layer，可以理解为 HTTP+SSL/TLS）在传输数据之前需要客户端（浏览器）与服务端（网站）之间进行一次握手，在握手过程中将确立双方加密传输数据的密码信息。HTTP 与 HTTPS 的主要区别可参考图 2-1 所示。

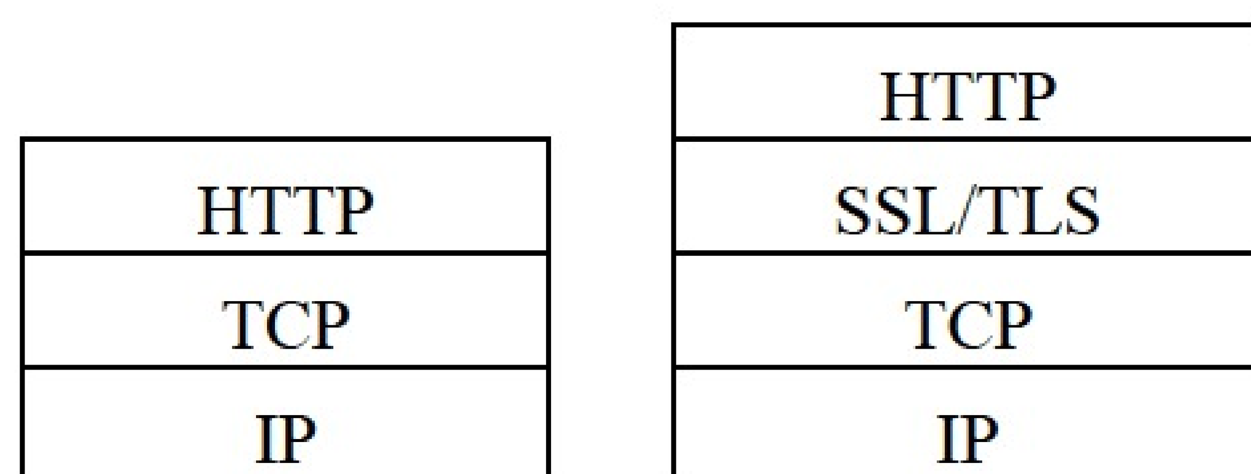


图 2-1 HTTP 与 HTTPS 的区别

HTTPS 的 SSL 中使用了非对称加密、对称加密以及 HASH 算法。握手过程的简单描述如下：

（1）浏览器将自己支持的一套加密规则发送给网站。

（2）网站从中选出一组加密算法与 HASH 算法，并将自己的身份信息以证书的形式发回给浏览器。证书里面包含网站地址、加密公钥以及证书的颁发机构等信息。

（3）获得网站证书之后浏览器要做以下工作：

① 验证证书的合法性（如颁发证书的机构是否合法、证书中包含的网站地址是否与正在访问的地址一致等），如果证书受信任，浏览器栏就会显示一个小锁头，否则会给出证书不受信任的提示。

② 如果证书受信任或者用户接受了不受信任的证书，浏览器就会生成一串随机数的密码，并用证书中提供的公钥加密。

③ 使用约定好的 HASH 计算握手消息，并使用生成的随机数对消息进行加密，最后将之前生成的所有信息发送给网站。

（4）网站接收浏览器发来的数据之后要做以下操作：

① 使用自己的私钥将信息解密并取出密码，使用密码解密浏览器发来的握手消息，并验证 HASH 是否与浏览器发来的一致。

② 使用密码加密一段握手消息，发送给浏览器。

（5）如果浏览器解密并计算握手消息的 HASH 与服务端发来的 HASH 一致，此时握手过程结束，之后所有的通信数据将使用之前浏览器生成的随机密码，并利用对称加密算法进行加密。

浏览器与网站互相发送加密的握手消息并验证，目的是保证双方都获得一致的密码，并且可以正常地加密、解密数据，为真正数据的传输做一次测试。另外，HTTPS 一般使用的加密与 HASH 算法如下。

（1）非对称加密算法：RSA、DSA/DSS。

（2）对称加密算法：AES、RC4、3DES。

（3）HASH 算法：MD5、SHA1、SHA256。

其中，非对称加密算法用于在握手过程中加密生成的密码，对称加密算法用于对真正传输的数据进行加密，而 HASH 算法用于验证数据的完整性。由于浏览器生成的密码是整个数据加密的关键，因此在传输的时候使用非对称加密算法对其加密。非对称加密算法会生成公钥和私钥，公钥只能用于加密数据，可以随意传输，而网站的私钥用于对数据进行解密，所以网站都会非常小心地保管自己的私钥，防止泄漏。

SSL 握手过程中有任何错误都会使加密连接断开，从而阻止隐私信息的传输，正是由于 HTTPS 非常安全，攻击者无法从中找到下手的地方，因此更多地采用假证书的手法来欺骗客户端，从而获取明文的信息。

2.2　请　求　头

请求头描述客户端向服务器发送请求时使用的协议类型、所使用的编码以及发送内容的长度等。客户端（浏览器）通过输入 URL 后确定等于做了一次向服务器的请求动作，在这个请求里面带有请求参数，请求头在网络爬虫中的作用是相当重要的一部分。检测请求头是常见的反爬虫策略，因为服务器会对请求头做一次检测来判断这次请求是人为的还是非人为的。为了形成一个良好的代码编写规范，无论网站是否做 Headers 反爬虫机制，最好每次发送请求都添加请求头。

请求头的参数如下。

（1）Accept：text/html,image/*（浏览器可以接收的文件类型）。

（2）Accept-Charset：ISO-8859-1（浏览器可以接收的编码类型）。

（3）Accept-Encoding：gzip,compress（浏览器可以接收的压缩编码类型）。

（4）Accept-Language：en-us,zh-cn（浏览器可以接收的语言和国家类型）。

（5）Host：请求的主机地址和端口。

（6）If-Modified-Since：Tue, 11 Jul 2000 18:23:51 GMT（某个页面的缓存时间）。

（7）Referer：请求来自于哪个页面的 URL。

（8）User-Agent：Mozilla/4.0（compatible，MSIE 5.5，Windows NT 5.0，浏览器相关信息）。

（9）Cookie：浏览器暂存服务器发送的信息。

（10）Connection：close(1.0)/Keep-Alive(1.1)（HTTP 请求版本的特点）。

（11）Date：Tue, 11 Jul 2000 18:23:51 GMT（请求网站的时间）。

一个标准的请求基本上都带有以上属性。在网络爬虫中，请求头一定要有 User-Agent，其他的属性可以根据实际需求添加，因为反爬虫通常检测请求头的 Referer 和 User-Agent，而 Cookie 不能添加到请求头。除此之外，还有一些比较特殊的请求头信息，如 Upgrade-Insecure-Requests（告诉服务器，浏览器可以处理 HTTPS 协议）、X-Requested-With（判断是否 Ajax 请求）等。

以下是 Python 里面一个完整的请求头，以字典格式生成，代码如下：

```
Headers = {
    'Accept': 'text/html,application/xhtml+xml,
              application/xml;q=0.9 ,*/*;q=0.8',
    'Accept-Language': 'zh-CN,zh;q=0.8',
    'Cache-Control': 'max-age=0',
    'User-Agent': ' Mozilla/5.0 (Windows NT 6.3;
             WOW64; rv:41.0) Gecko/20100101 Firefox/41.0',
    'Connection': 'keep-alive',
        'Referer': 'https://movie.douban.com/'}
```

2.3 Cookies

Cookies 也可以称为 Cookie，指某些网站为了辨别用户身份、进行 Session 跟踪而储存在用户本地终端上的数据。

一个 Cookies 就是存储在用户主机浏览器中的文本文件。Cookies 是纯文本形式，它们不包含任何可执行代码。服务器告诉浏览器将这些信息存储，并且每个请求中都将该信息返回到服务器。服务器之后可以利用这些信息来标识用户。多数需要登录的网站通常会在用户登录后将用户信息写入 Cookies，只要这个 Cookies 存在并且合法，就可以自由地浏览这个网站的所有站点。Cookies 只是包含数据，就其本身而言并不有害。

服务器可以利用 Cookies 包含的信息判断在 HTTP 传输中的状态。Cookies 最典型的应用是判定注册用户是否已经登录网站和保留用户信息以便简化登录手续。

一般 Cookies 所具有的属性如下。

- Domain：域，表示当前 Cookies 属于哪个域或子域下面。
- Path：表示 Cookies 的所属路径。
- Expire Time/Max-Age：表示 Cookies 的有效期。
- Secure：表示该 Cookies 只能用 HTTPS 传输。
- Httponly：表示此 Cookies 必须用 HTTP 或 HTTPS 传输。
- HasKeys：通过该值指示 Cookie 是否含有子键，返回一个 bool 值。
- Name：表示 Cookie 的名称。
- Value：单个 Cookie 的值。
- Values：单个 Cookie 所包含的键值对的集合。

Cookies 的优点如下。

（1）极高的扩展性和可用性。
（2）通过良好地编程控制保存在 Cookie 中的 Session 对象的大小。
（3）通过加密和安全传输技术（SSL）减少 Cookie 被破解的可能性。
（4）只在 Cookie 中存放不敏感数据，即使被盗也不会有重大损失。
（5）可控制 Cookie 的生命期，使之不会永远有效。

Cookies 的缺点如下。

（1）Cookie 数量和长度的限制。每个 domain 最多只能有 20 条 Cookie，每个 Cookie 长度不能超过 4KB，否则会被截掉。

（2）安全性问题。如果 Cookie 被拦截，就有可能被取得所有的 Session 信息。

（3）某些状态不可保存在客户端。例如，为了防止重复提交表单，需要在服务器端保存一个计数器。如果把这个计数器保存在客户端，那么它起不到任何作用。

2.4　HTML

HTML 是超文本标记语言，标准通用标记语言下的一个应用。“超文本”就是指页面内可以包含图片、链接，甚至音乐、程序等非文字元素。超文本标记语言的结构包括“头”部分（Head）和“主体”部分（Body），其中“头”部分提供关于网页的信息，“主体”部分提供网页的具体内容。

爬虫开发对 HTML 的要求是能看懂 HTML 各个标签的含义，了解标签的属性作用以及整个 HTML 布局设计。下面来看一个简单的 HTML 文档的结构：

```
<!DOCTYPE html> # 声明为 HTML5 文档
<html># 元素是 HTML 页面的根元素
<head># 元素包含了文档的元（meta）数据
<meta charset="utf-8"># 元素可提供有关页面的元信息（meta-information），主要是描述和关键词
<title>Python</title># 元素描述了文档的标题
</head>
<body> # 元素包含了可见的页面内容
<h1>我的第一个标题</h1> # 定义一个标题
<p>我的第一个段落。</p> # 元素定义一个段落
</body>
</html>
```

一个完整的网页必定以<html></html>为开头和结尾，整个 HTML 可分为两部分：

（1）<head></head>，主要是对网页的描述、图片和 JavaScript 的引用。<head> 元素包含所有的头部标签元素。在 <head>元素中可以插入脚本（scripts）、样式文件（CSS）及各种 meta 信息。该区域可添加的元素标签有<title>、<style>、<meta>、<link>、<script>、<noscript>和<base>。

（2）<body></body>是网页信息的主要载体。该标签下还可以包含很多类别的标签，不同的标签有不同的作用，标签以<>开头，以</>结尾，<>和</>之间的内容是标签的值和属性，每个标签之间可以是相互独立的，也可以是嵌套、层层递进的关系。

根据这两个组成部分就能很容易地分析整个网页的布局。其中，<body></body>是整个 HTML 的重点部分，通过示例讲述如何分析<body></body>：

```
<body>
<h1>我的第一个标题</h1>
<div>
<p> Python</p>
</div>
<h2>
<p>
<a> Python</a>
</p>
</h2>
</body>
```

上述例子分析如下：

（1）<h1>和<div>是两个不相关的标签，两个标签是相互独立的。

（2）<div>和<p>是嵌套关系，<p>的上一级标签是<div>。

（3）<h1>和<p>这两个标签是毫无关系的。

（4）<h2>标签包含一个<p>标签，<p>标签再包含一个<a>标签，一个标签可以包含多个标签在其中。

除上述示例的标签之外，大部分标签都可以在<body></body>中添加，常用的标签如表 2-1 所示。

表 2-1　HTML 常用的标签

HTML 标签	中文释义
Img	图片
A	锚
Strong	加重（文本）
Em	强调（文本）
I	斜体字
B	粗体（文本）
Br	换行
Div	分隔
Span	范围
Ol	排序列表
Ul	不排序列表
Li	列表项目
Dl	定义列表
h1~h6	标题 1 到标题 6
P	段落
Tr	表格中的一行
Th	表格中的表头
Td	表格中的一个单元格

2.5　JavaScript

JavaScript 是一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类型。它的解释器被称为 JavaScript 引擎，为浏览器的一部分，广泛用于客户端的脚本语言，最早是在 HTML 网页上使用的，用来给 HTML 网页增加动态功能。

JavaScript 脚本语言同其他语言一样，有自身的基本数据类型、表达式和算术运算符及程序的基本框架。JavaScript 提供了 4 种基本的数据类型和两种特殊的数据类型用来处理数据和文字。而变量提供存放信息的地方，表达式则可以完成较复杂的信息处理。

有时候分析网站需要理解某些 JavaScript 的功能，如某些特殊的数据会存放在 JavaScript 中。以 12306 全国站点为例，如图 2-2 所示。

```
不安全 | https://kyfw.12306.cn/otn/resources/js/framework/station_name.js?station_version=1.9031

var station_names =' @bjb|北京北|VAP|beijingbei|bjb|0@bjd|北京东|BOP|beijingdong|bjd|1@bji|北京|BJP|beijing|bj|2@bjn|北京南|V
南|IZQ|guangzhounan|gzn|5@cqb|重庆北|CUW|chongqingbei|cqb|6@cqi|重庆|CQW|chongqing|cq|7@cqn|重庆南|CRW|chongqingnan|cqn|8@gz
海南|SNH|shanghainan|shn|11@shq|上海虹桥|AOH|shanghaihongqiao|shhq|12@shx|上海西|SXH|shanghaixi|shx|13@tjb|天津北|TBP|tianji
南|TIP|tianjinnan|tjn|16@tjx|天津西|TXP|tianjinxi|tjx|17@cch|长春|CCT|changchun|cc|18@ccn|长春南|CET|changchunnan|ccn|19@ccx
东|ICW|chengdudong|cdd|21@cdn|成都南|CNW|chengdunan|cdn|22@cdu|成都|CDW|chengdu|cd|23@csh|长沙|CSQ|changsha|cs|24@csn|长沙南
南|FYS|fuzhounan|fzn|27@gya|贵阳|GIW|guiyang|gy|28@gzh|广州|GZQ|guangzhou|gz|29@gzx|广州西|GXQ|guangzhouxi|gzx|30@heb|哈尔滨
西|VAB|haerbinxi|hebx|33@hfe|合肥|HFH|hefei|hf|34@hfx|合肥西|HTH|hefeixi|hfx|35@hhd|呼和浩特东|NDC|huhehaotedong|hhhtd|36@hh
东|KEQ|haikoudong|hkd|38@hkd|海口东|HMQ|haikoudong|hkd|39@hko|海口|VUQ|haikou|hk|40@hzd|杭州东|HGH|hangzhoudong|hzd|41@hzh|杭
南|JNK|jinan|jn|44@jnd|济南东|JAK|jinandong|jnd|45@jnx|济南西|JGK|jinanxi|jnx|46@kmi|昆明|KMM|kunming|km|47@kmx|昆明西|KXM|k
东|LVJ|lanzhoudong|lzd|50@lzh|兰州|LZJ|lanzhou|lz|51@lzx|兰州西|LAJ|lanzhouxi|lzx|52@nch|南昌|NCG|nanchang|nc|53@nji|南京|NJ
```

图 2-2　12306 站点信息

从图 2-2 中可以看到，不同的站点有对应的英文字母，代表站点的编码信息，JavaScript 存储数据主要使用变量的形式。

JavaScript 还能根据用户触发某些事件对用户的操作进行加工处理。例如用户登录信息设置加密处理，原理是先对用户提交的信息做加密处理，然后发送请求到服务器，这一系列事件由 JavaScript 独立完成。要用爬虫实现该功能，就要分析 JavaScript 如何执行整个用户登录过程。

下面通过分析一个简单的例子，来进一步了解 JavaScript 事件的触发原理：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<script> <!-- //JavaScript 代码 -→
function validateForm() {
    var x = document.forms["myForm"]["fname"].value;
    if (x == null || x == "") {
        alert("需要输入名字。");
        return false;
    }
    else{alert("你的名字提交成功")}
}
</script>
</head>
<body>
<!-- //html 表单，当点击按钮"提交"后会触发 onsubmit 这个事件，执行 JavaScript 代码 -→
<form name="myForm" action="" onsubmit="return validateForm()" method="post">
名字: <input type="text" name="fname">
<input type="submit" value="提交">
</form>
</body>
</html>
```

JavaScript 事件的触发过程：

（1）HTML 根据<form></form>标签相应地生成一个表单。

（2）当用户在表单输入内容后，单击提交按钮，就会触发<form></form>表单里所指向的 validateForm()方法，执行相应的 JavaScript 代码。

（3）validateForm()会判断输入的值是否为空。如果输入的值为空，就提示输入名字；若输入的值不为空，则提示“提交成功”。

2.6 JSON

JSON（JavaScript Object Notation，JavaScript 对象标记）是一种轻量级的数据交换格式，采用完全独立于编程语言的文本格式来存储和表示数据。简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言，易于阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

在 JavaScript 语言中，一切都是对象。因此，任何支持的类型都可以通过 JSON 来表示，例如字符串、数字、对象、数组等。JSON 格式说明如下：

（1）对象表示为键值对。
（2）数据由逗号分隔。
（3）花括号保存对象。
（4）方括号保存数组。

JSON 的书写格式是：键/值对，包括字段名称（字符串），后面写一个冒号，然后是值。例如“name”:“Tom”，等价于 JavaScript 语句：name = “Tom”

JSON 的值可以是数字（整数或浮点数）、字符串、逻辑值（True 或 False）、数组（在方括号中）、对象（在花括号中）和 Null。

例子如下：

```
MyJSon = {
"name": "Python",
"address" : { "province" : "广东" , "city" : "广州"}
}
```

JSON 的格式是用花括号表示的，代码 MyJSon 里包含两个属性，分别是 name 和 address。name 的值是“Python”；address 的值是嵌套新的 JSON，里面包含 province 和 city 属性，值为“广东”和“广州”。

一个 JSON 里可以嵌套多个 JSON，也可以嵌套 JSON 数组，都是以键-值的形式表现。在数据结构上，JSON 与 Python 里的字典非常相似。

2.7 Ajax

Ajax 不是一种新的编程语言，而是一种用于创建更好、更快以及交互性更强的 Web 应用程序的技术。使用 JavaScript 向服务器提出请求并处理响应而不阻塞用户，核心对象是 XMLHTTPRequest。通过这个对象，JavaScript 可在不重载页面的情况下与 Web 服务器交换数据，即在不需要刷新页面的情况下就可以产生局部刷新的效果。

Ajax 在浏览器与 Web 服务器之间使用异步数据传输（HTTP 请求），这样就可以使网页从服务器请求少量的信息，而不是整个页面。

JavaScript、XML、HTML 与 CSS 在 Ajax 中使用的 Web 标准已被良好定义，并被所有的主流浏览器支持，Ajax 应用程序独立于浏览器和平台。

Web 应用程序比桌面应用程序有优势，能够涉及广大的用户，更易安装及维护，也更易开发。

判断网页数据是否使用 Ajax 最简单的方法是：触发事件之后，判断网页是否发生刷新状态。如果网页没有发生刷新，数据就自动生成，说明数据的加载是通过 Ajax 生成并渲染到网页上的；反之，数据是通过服务器后台生成并加载的。

两种数据加载渲染方式分别由前端和后端完成，实现的方式和原理也不同。判断数据加载方式是爬虫开发必备的基本技能之一，正确地判断数据加载方式才能找到数据来源的渠道，最终才能找到抓取的目标。

2.8 本章小结

本章主要介绍了与编写爬虫程序相关的 Web 前端开发技术。

前端开发技术是爬虫开发人员必备技能之一，也是编写爬虫程序的基础。前端技术的主要作用是分析各类网站的设计架构，以便有针对性地编写爬虫脚本。从整个爬虫开发周期来看，分析网站架构是最为耗时的一环，也是爬虫开发的核心之一，可以说，爬虫的开发都是基于网站的分析为前提。

关于前端开发技术，读者应重点掌握以下内容。

- HTTP 与 HTTPS：互联网上应用最为广泛的一种网络协议。目前所有网站开发都基于该协议，也是网站的实现原理。
- 请求头：基于 HTTP 与 HTTPS 协议实现，其作用是在通信之间实现信息传递。熟知各种请求类型，对爬虫中编写请求头有指导性作用。
- Cookies：存储在用户主机浏览器中的文本文件，主要让服务器识别各个用户身份信息。
- HTML：服务器返回的网页内容，一般由服务器后台生成。网站大部分数据来源于此，熟悉 HTML 布局和各个标签的作用，有利于数据抓取和清洗。
- JavaScript：主要实现网页的动态功能及用户交互。要懂得分析 JavaScript 代码，尤其是数据加密处理。
- JSON：表示一个 JavaScript 对象的信息，本质是一个特殊的字符串。
- Ajax：主要是前端数据加载和渲染技术，其响应内容大部分以 JSON 格式为主。

第 3 章

Chrome 分析网站

3.1　Chrome 开发工具

浏览器是从事编程开发人员必备的开发工具。世界上五大主流浏览器分别是：IE、Opera、Google Chrome、Safari 和 Firefox，其中 Chrome 和 Firefox 是编程开发人员的首选，主要是两者运行速度、扩展性和用户体验都符合开发人员所需。

本书选择 Chrome 作为分析网站的工具，因为其简洁、速度快（无论是启动速度、页面解析速度还是 JavaScript 执行速度），对 HTML5 和 CSS3 的支持也比较完善。

以分析豆瓣电影为例，先打开 Chrome 浏览器，进入豆瓣电影网页（https://movie.douban.com/）。单击 Chrome 的开发者工具（快捷键：F12），如图 3-1 所示。

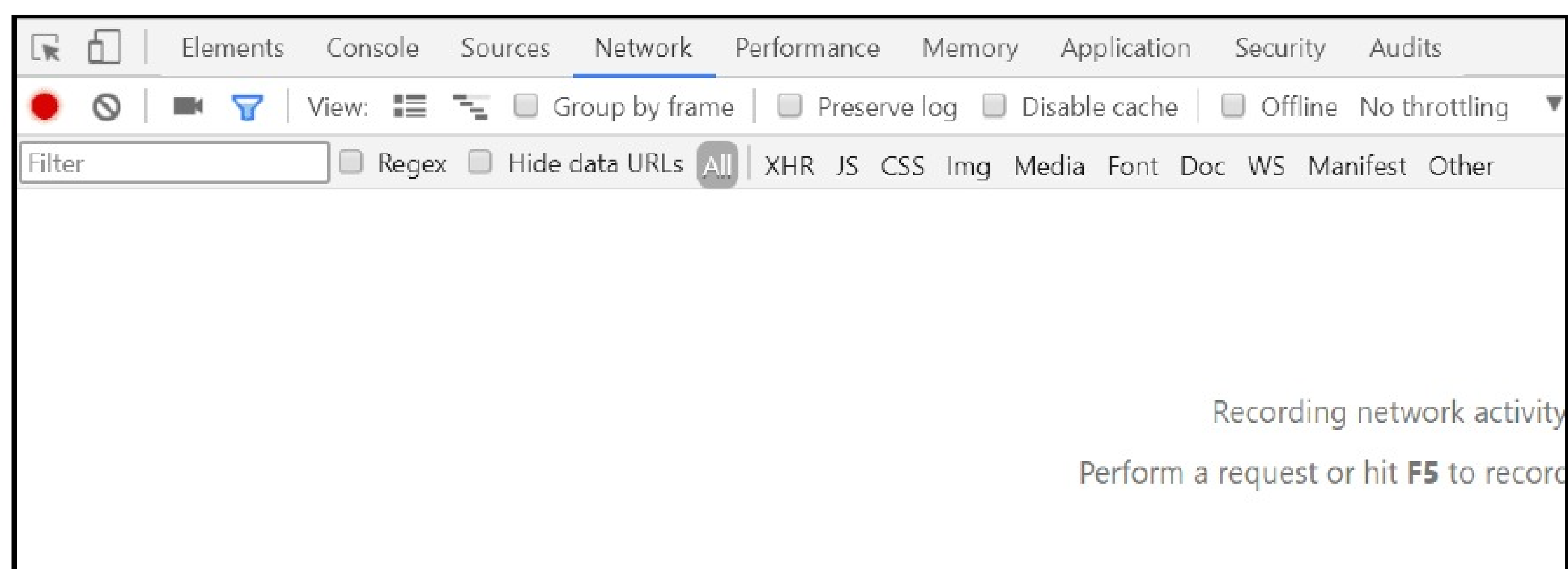


图 3-1　开发者模式

还可以通过在网页上右击，选择“检查”，或者按 Ctrl+Shift+I 组合键，如图 3-2 所示，打开开发者工具界面。

开发者工具的界面共有 9 个标签页，分别是：Elements、Console、Sources、Network、Performance、Memory、Application、Security 和 Audits。

Chrome 开发者工具以 Web 调试为主，如果用于爬虫分析，熟练掌握 Elements 和 Network 标

签就能满足大部分的爬虫需求。其中，Network 是核心部分。

返回(B)	Alt+向左箭头
前进(F)	Alt+向右箭头
重新加载(R)	Ctrl+R
另存为(A)...	Ctrl+S
打印(P)...	Ctrl+P
投射(C)...	
翻成中文（简体）(T)	
查看网页源代码(V)	Ctrl+U
检查(N)	Ctrl+Shift+I

图 3-2　开发者模式

3.2　Elements 标签

在 Elements 标签中允许从浏览器的角度看页面，也就是说可以看到 Chrome 渲染页面所需要的 HTML、CSS 和 DOM（Document Object Model）对象。此外，还可以编辑内容更改页面显示效果，如图 3-3 所示。

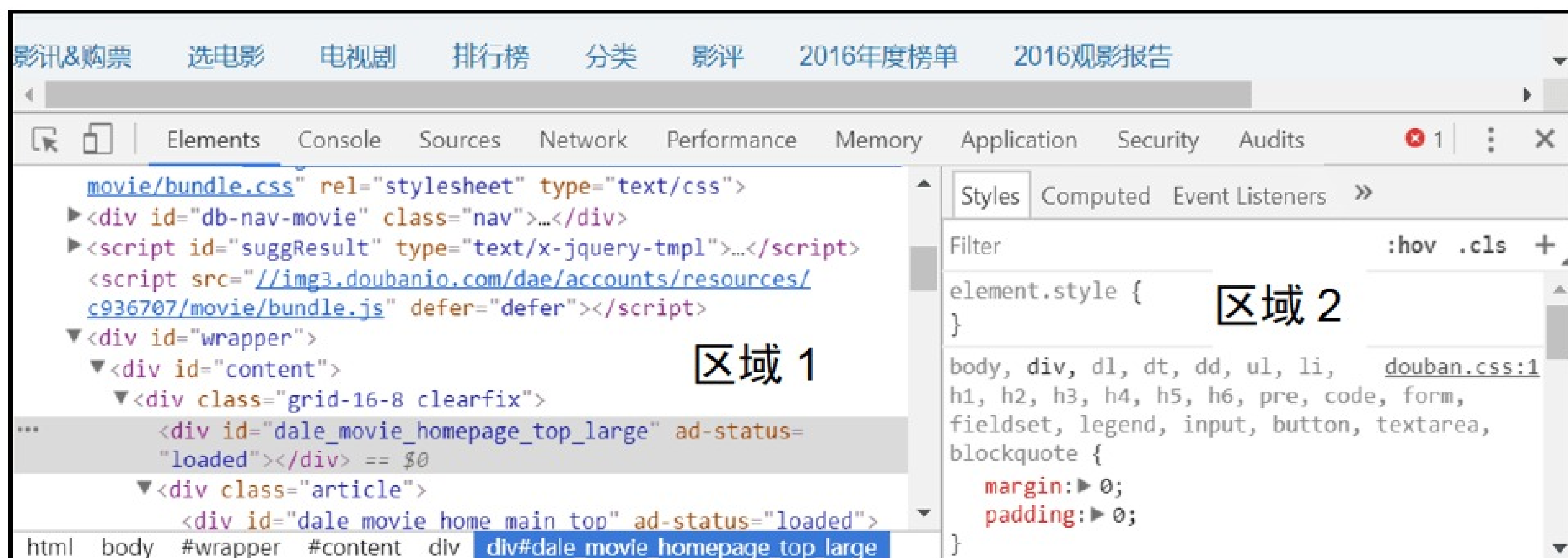


图 3-3　Elements 标签

图 3-3 中，Elements 标签最左边的按钮用于快速查找网页元素，单击该按钮后，在网页上某一处单击，就会自动显示并选中该元素在 HTML 里的位置。

Elements 标签分成两部分，分别在图 3-3 中标为区域 1 和区域 2，两个区域相辅相成。区域 1 显示整个网页的 HTML 信息，单击选中某一行内容的时候，区域 2 的 Styles 标签会显示当前单击选中内容的 CSS 样式，并可对元素的 CSS 进行查看与编辑修改。Computed 显示当前选中的边距属性、边框属性，用图像显示一个整体效果。Event Listeners 是整个网页事件触发的 JavaScript，如图 3-4 所示。

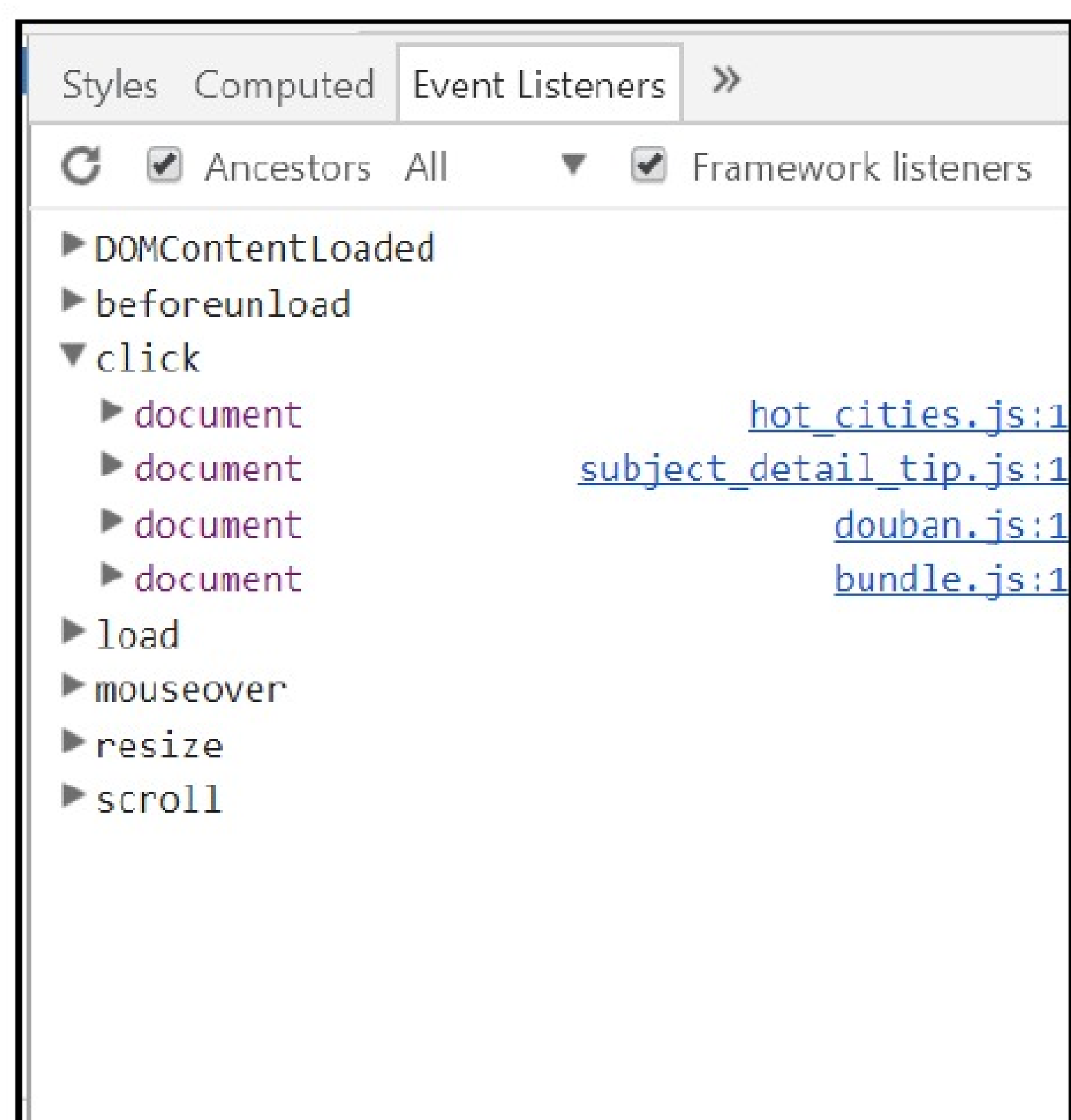


图 3-4　Event Listeners

通过单击 Event Listeners 下的某个 JavaScript 会自动跳转到 Sources 标签，显示当前 JavaScript 的源码，这个功能可快速找到 JavaScript 代码所在的位置，对分析 JavaScript 起到快速定位作用。

3.3　Network 标签

在 Network 标签中可以看到页面向服务器请求的信息、请求的大小以及加载请求花费的时间。从发起网页页面请求 Request 后分析 HTTP 请求得到各个请求信息（包括状态、类型、大小、所用时间、Request 和 Response 等）。Network 标签的结构组成如图 3-5 所示。

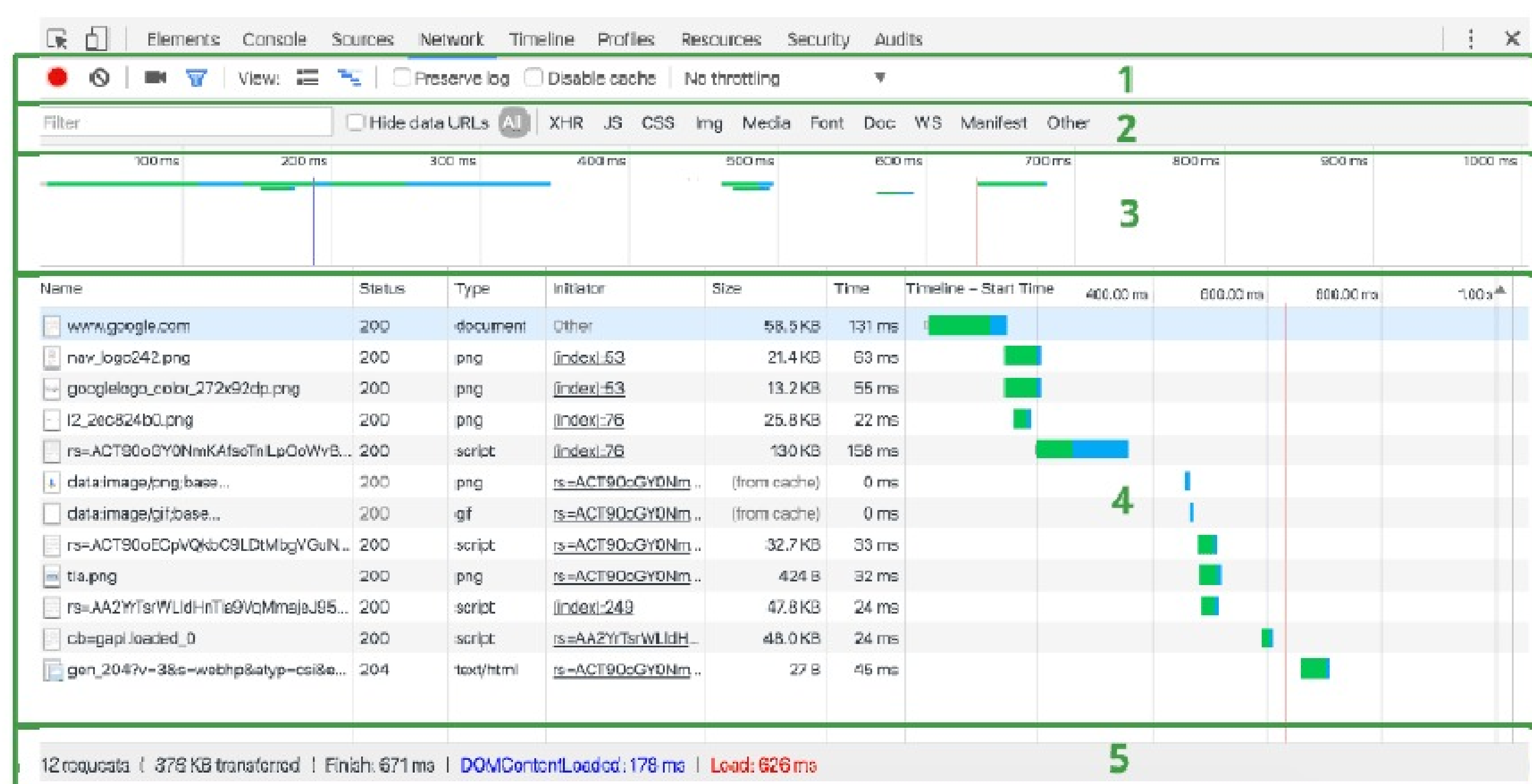


图 3-5　Network 标签

Network 标签主要包括以下 5 个区域。

- Controls：控制 Network 的外观和功能。
- Filters：控制 Requests Table 具体显示哪些内容。

  - ➢ All：返回当前页面全部加载的信息，就是一个网页全部所需要的代码、图片等请求。
  - ➢ XHR：筛选 Ajax 的请求链接信息，前面讲过 Ajax 核心对象 XMLHTTPRequest，XHR 取于 XMLHTTPRequest 的缩写。
  - ➢ JS：主要筛选 JavaScript 文件。
  - ➢ CSS：主要是 CSS 样式内容。
  - ➢ Img：是网页加载的图片，爬取图片的 URL 都可以在这里找到。
  - ➢ Media：是网页加载的媒体文件，如 MP3、RMVB 等音频视频文件资源。
  - ➢ Doc：是 HTML 文件，主要用于响应当前 URL 的网页内容。
- Overview：显示获取到请求的时间轴信息，主要是对每个请求信息在服务器的响应时间进行记录。这个主要是为网站开发优化方面提供数据参考，这里不做详细介绍。
- Requests Table：按前后顺序显示所有捕捉的请求信息，单击请求信息可以查看该详细信息。
- Summary：显示总的请求数、数据传输量、加载时间信息。

5 个区域中，Requests Table 是核心部分，主要作用是记录每个请求信息。但每次网站出现刷新时，请求列表都会清空并记录最新的请求信息，如用户登录后发生 304 跳转，就会清空跳转之前的请求信息并捕捉跳转后的请求信息。

对于每条请求信息，可以单击查看该请求的详细信息，如图 3-6 所示。

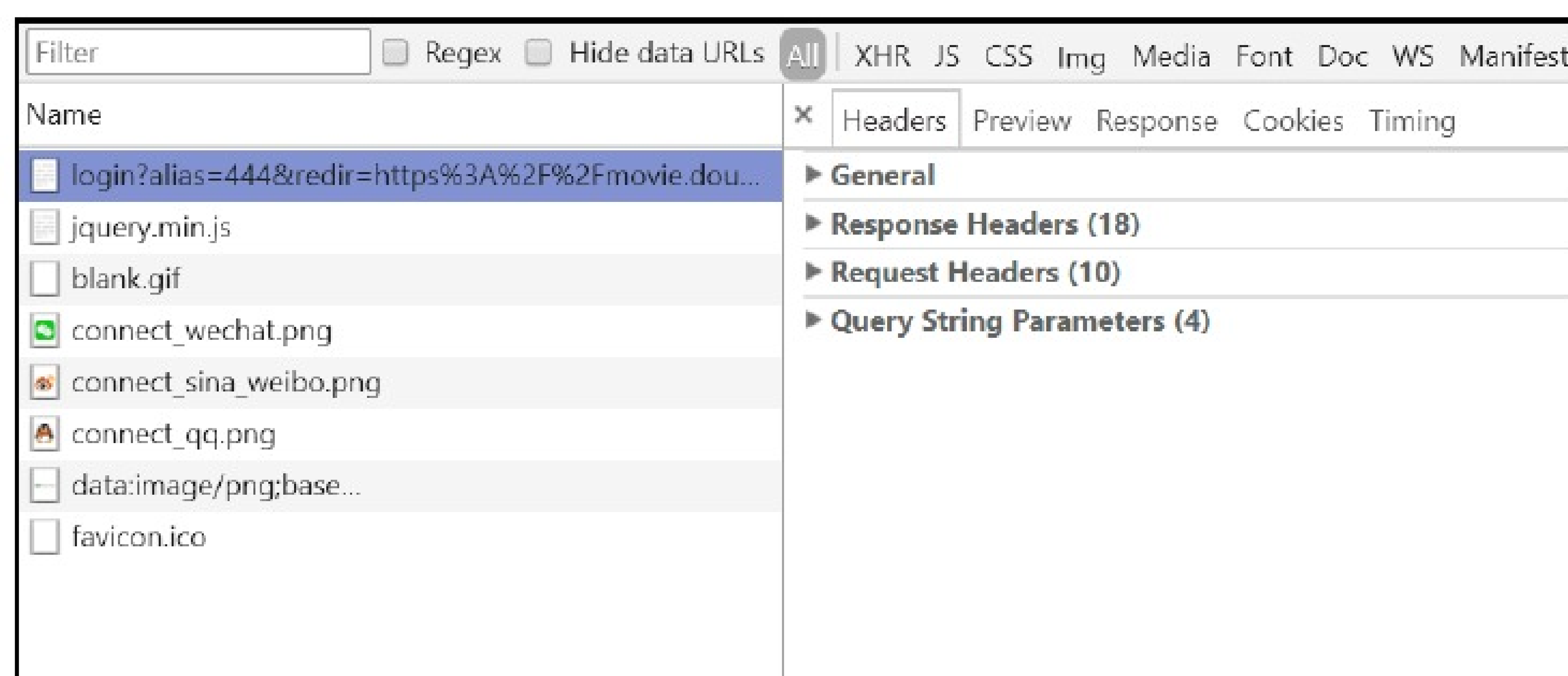


图 3-6 请求信息

每条请求信息划分为以下 5 个标签。

- Headers：该请求的 HTTP 头信息。
- Preview：根据所选择的请求类型（JSON、图片、文本）显示相应的预览。
- Response：显示 HTTP 的 Response 信息。
- Cookies：显示 HTTP 的 Request 和 Response 过程中的 Cookies 信息。
- Timing：显示请求在整个生命周期中各部分花费的时间。

常用的标签有 Headers、Preview 和 Response。Headers 用于获取请求链接、请求头和请求参数；Preview 和 Response 用于显示服务器返回的响应内容。

Headers 标签划分为以下 4 部分。

- General：记录请求链接、请求方式和请求状态码。
- Response Headers：服务器端的响应头，其参数说明如下。
  - ➢ Cache-Control：指定缓存机制，优先级大于 Last-Modified。

  - Connection：包含很多标签列表，其中最常见的是 Keep-Alive 和 Close，分别用于向服务器请求保持 TCP 连接和断开 TCP 连接。
  - Content-Encoding：服务器通过这个头告诉浏览器数据的压缩格式。
  - Content-Length：服务器通过这个头告诉浏览器回送数据的长度。
  - Content-Type：服务器通过这个头告诉浏览器回送数据的类型。
  - Date：当前时间值。
  - Keep-Alive：在 Connection 为 Keep-Alive 时，该字段才有用，用来说明服务器估计保留连接的时间和允许后续几个请求复用这个保持着的连接。
  - Server：服务器通过这个头告诉浏览器服务器的类型。
  - Vary：明确告知缓存服务器按照 Accept-Encoding 字段的内容分别缓存不同的版本。
- Request Headers：用户的请求头。其参数说明如下。
  - Accept：告诉服务器客户端支持的数据类型。
  - Accept-Encoding：告诉服务器客户端支持的数据压缩格式。
  - Accept-Charset：可接受的内容编码 UTF-8。
  - Cache-Control：缓存控制，服务器控制浏览器要不要缓存数据。
  - Connection：处理完这次请求后，是断开连接还是保持连接。
  - Cookie：客户可通过 Cookie 向服务器发送数据，让服务器识别不同的客户端。
  - Host：访问的主机名。
  - Referer：包含一个 URL，用户从该 URL 代表的页面出发访问当前请求的页面，当浏览器向 Web 服务器发送请求的时候，一般会带上 Referer，告诉服务器请求是从哪个页面 URL 过来的，服务器借此可以获得一些信息用于处理。
  - User-Agent：中文名为用户代理，简称 UA，是一个特殊字符串头，使得服务器能够识别客户使用的操作系统及版本、CPU 类型、浏览器及版本、浏览器渲染引擎、浏览器语言、浏览器插件等。
- Query String Parameters：请求参数。主要是将参数按照一定的形式（GET 和 POST）传递给服务器，服务器通过接收其参数进行相应的响应，这是客户端和服务端进行数据交互的主要方式之一。

Headers 标签的内容看起来很多，但在实际使用过程中，爬虫开发人员只需关心请求链接、请求方式、请求头和请求参数的内容即可。而 Preview 和 Response 是服务器返回的结果，两者之间对不同类型的响应结果有不同的显示方式：

（1）如果返回的结果是图片，那么 Preview 表示可显示图片内容，Response 表示无法显示。

（2）如果返回的是 HTML 或 JSON，那么两者皆能显示，但在格式上可能会存在细微的差异。

3.4 分析 QQ 音乐

现在以 QQ 音乐某一歌手页面的分析为例（y.qq.com/n/yqq/singer/0025NhlN2yWrP4.html）讲述如何使用 Chrome 开发者工具分析网站，如图 3-7 所示。

图 3-7　歌手信息

从图 3-7 中可以看到，在 Network 标签下捕捉到很多请求信息，请求类型有 document、png、font 和 script 等，分别对应 HTML 文件、图片、字体格式和 JavaScript 脚本。

单击“Filters”下的 Doc 标签（Doc 是当前网页的 HTML 文件），发现有两个请求信息，分别是“0025NhlN2yWrP4.html”和“xhr_proxy_utf8.html”。从请求的命名可以看出，第一个请求与网站的 URL 是一致的。再查看“0025NhlN2yWrP4.html”的响应内容（Preview 标签），可以使用“Ctrl+F”快速查找歌曲信息，如图 3-8 所示。

图 3-8　快速查找歌曲信息

在 Doc 中虽能找到歌曲名、专辑和时长，但无法找到更多的歌曲信息。歌曲信息有可能是由其他方式生成的，网站数据生成只有前端（Ajax 或 JSONP）和后端（服务器）两种方式。从图 3-8 返回的结果来看，数据不可能是从后端生成的，那么就可能是由前端加载生成的。

提　示

JSONP（JSON With Padding）是 JSON 的一种“使用模式”，可用于解决主流浏览器的跨域数据访问问题。

前端加载的数据有可能记录在 Chrome 开发者工具的“XHR”或“JS”中，分别查看两个标签里面的请求信息，最终发现歌曲信息存放在 JS 下的某个请求中，如图 3-9 和图 3-10 所示。

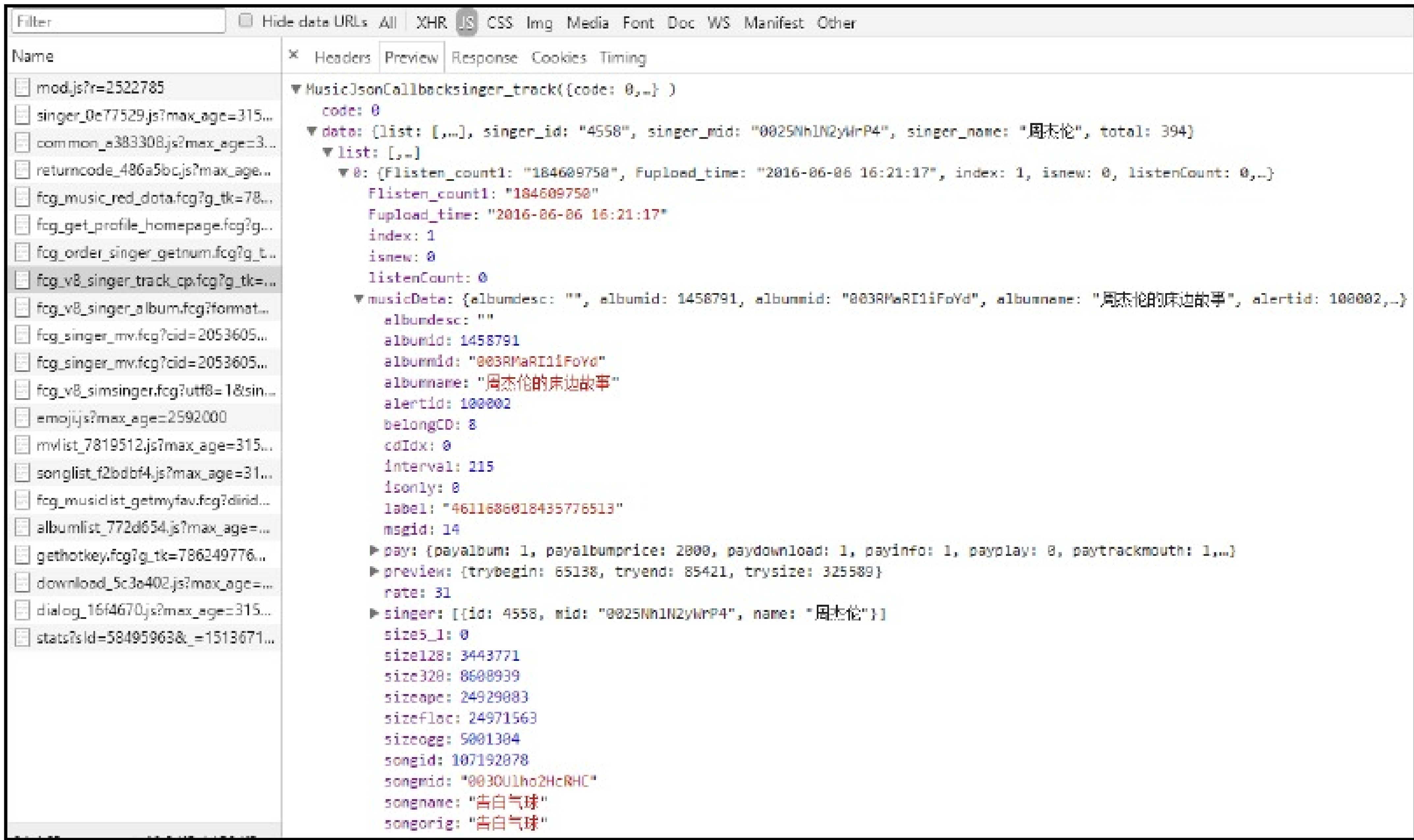


图 3-9　响应内容

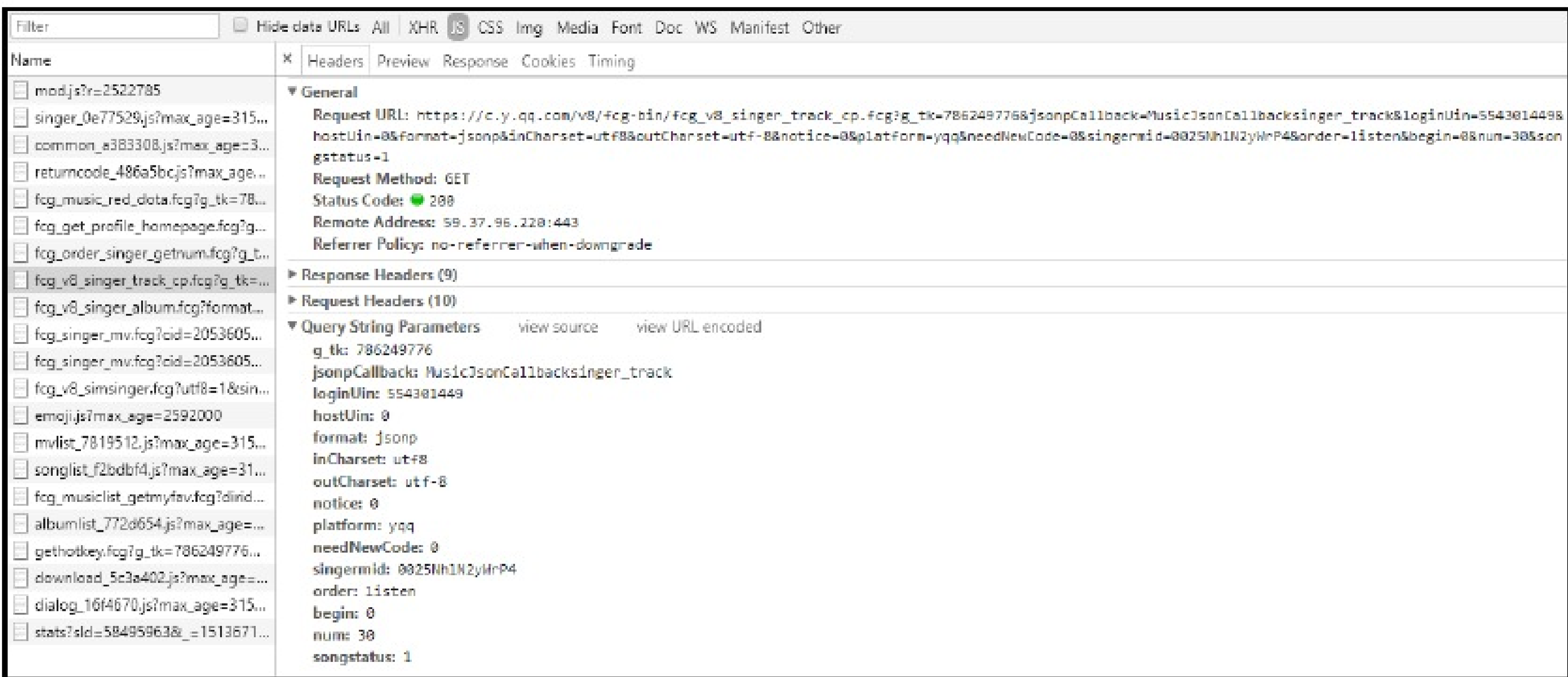


图 3-10　请求信息

从图 3-9 和图 3-10 分析得知，请求方式是 GET，Query String Parameters 是记录该请求的参数。因为请求方式是 GET，所以请求参数也可以在请求链接上找到。

再看请求参数，大部分请求参数是可以明确知道的，唯独参数 singermid 无法确定。从参数的命名来看，这应该是歌手的 ID 信息，也是网站用于标记歌手唯一的属性，所以要获取歌手的 singermid 可能需要从其他请求上获取。

根据上述例子，可简单总结出分析网站的步骤如下：

步骤01 找出数据来源，大部分数据来源于 Doc、XHR 和 JS 标签。

步骤02 找到数据所在的请求，分析其请求链接、请求方式和请求参数。

步骤03 查找并确定请求参数来源。有时候某些请求参数是通过另外的请求生成的，比如请求 A 的参数 id 是通过请求 B 所生成的，那么要获取请求 A 的数据，就要先获取请求 B 的数据作为 A 的请求参数。

3.5 本章小结

Chrome 开发者工具的主要作用是进行 Web 开发调试，对于爬虫开发人员来说，应该熟练掌握 Elements、Console 和 Network。其中 Network 是核心部分，百分之九十的网站分析都在 Network 上完成，读者对 Network 上的各个功能和作用要理解掌握，并懂得如何使用 Chrome 分析网站的请求信息。

一般分析网站最主要的是找到数据的来源，确定数据来源就能确定数据生成的具体方法。总结归纳分析网站的步骤如下：

（1）找出数据来源，大部分数据来源于 Doc、XHR 和 JS 标签。

（2）找到数据所在的请求，分析其请求链接、请求方式和请求参数。

（3）查找并确定请求参数来源。有时候某些请求参数是通过另外的请求生成的，比如请求 A 的参数 id 是通过请求 B 所生成的，那么要获取请求 A 的数据，就要先获取请求 B 的数据作为 A 的请求参数。

上述分析步骤适用于大部分网站，但每个网站都有自身的设计特点，不能一概而论。此方法更多的是起到指导性作用，遇到具体的问题还是要具体分析。

第 4 章

Fiddler 抓包

4.1 Fiddler 介绍

Fiddler 是一款非常流行并且实用的 HTTP 抓包工具，原理是在电脑上开启一个 HTTP 代理服务器，然后转发所有的 HTTP 请求和响应。因此，比一般的浏览器自带的抓包工具（开发者工具）要好用得多。不仅如此，还可以支持请求重放一些高级功能，也可以支持对手机应用进行 HTTP 抓包。

Fiddler 是用 C#开发的工具，包含一个简单却功能强大的基于 JScript .NET 事件的脚本子系统，灵活性非常棒，可以支持众多的 HTTP 调试任务，并且能够使用.net 框架语言进行扩展。

此外，还支持断点调试技术，当请求或响应属性能够跟目标的标准相匹配时，Fiddler 就能够暂停 HTTP 通信，并且允许修改请求和响应。这种功能对于安全测试非常有用，当然也可以用来做一般的功能测试。

4.2 Fiddler 安装配置

Fiddler 在 Windows 下可直接使用 exe 安装包安装，安装包可在官方网站下载（https://www.telerik.com/download/fiddler）。

完成安装后，在安装目录下双击打开应用程序 Fiddler.exe，可看到 Fiddler 用户界面，如图 4-1 所示。

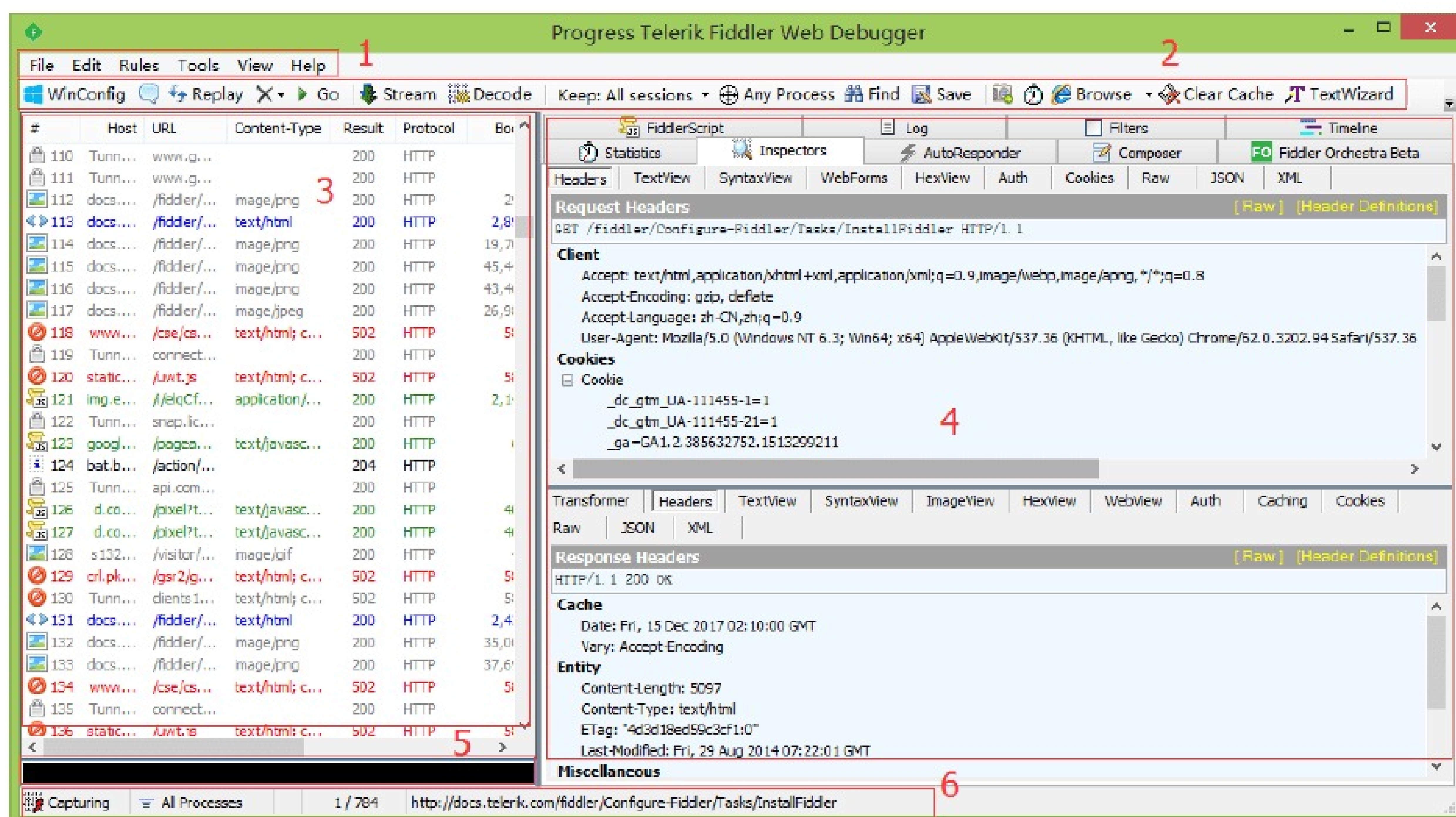


图 4-1　Fiddler 用户界面

Fiddler 用户界面主要包括下面 6 个部分：

（1）图中标注 1 为 Main Menu（主菜单），作用于整个 Fiddler 相关配置。
（2）图中标注 2 为 Toolbar（工具栏），主要对 Web Session 操作处理。
（3）图中标注 3 为 Web Session（列表），显示已抓取的 HTTP 请求信息。
（4）图中标注 4 为 View（选项视图），显示每条 HTTP 的详细信息。
（5）图中标注 5 为 Quickexec（命令行），通过特定的条件快速找到符合条件的 HTTP 请求。
（6）图中标注 6 为 Status bar（状态栏），显示当前状态信息。

打开 Fiddler 之后，由于 HTTPS 协议的特殊性，还需要配置 Fiddler。了解 Fiddler 抓取 HTTPS 协议的原理才能更好地理解如何对 Fiddler 进行配置，原理如图 4-2 所示。

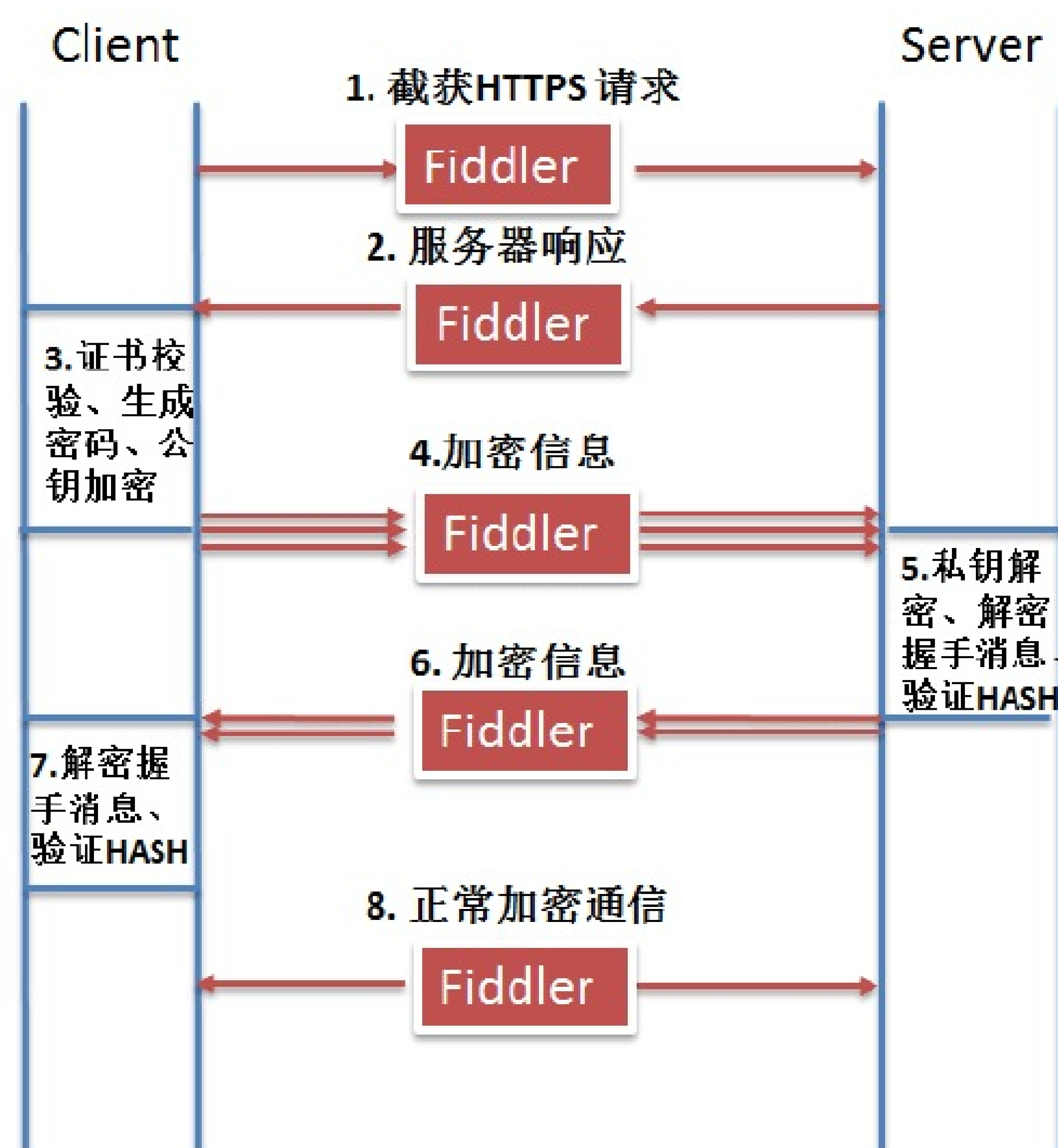


图 4-2　Fiddler 抓取 HTTPS 的原理

Fiddler 抓取 HTTPS 协议充当的角色：

（1）服务器→客户端：Fiddler 接收到服务器发送的密文，用对称密钥解开，获得服务器发送的明文。再次加密，发送给客户端。

（2）客户端→服务器端：客户端用对称密钥加密，被 Fiddler 截获后，解密获得明文。再次加密，发送给服务器端。由于 Fiddler 一直拥有通信用对称密钥 enc_key，因此在整个 HTTPS 通信过程中信息对其透明。

配置 Fiddler，使其能够抓取 HTTPS 请求信息，方法如下：

步骤01 对 Fiddler 进行设置：打开 Main Menu→Tools→Fiddler Options→HTTPS。

步骤02 勾选 HTTPS 里的选项，然后单击 Actions→Trust Root Certificate，完成证书验证，如图 4-3 所示。

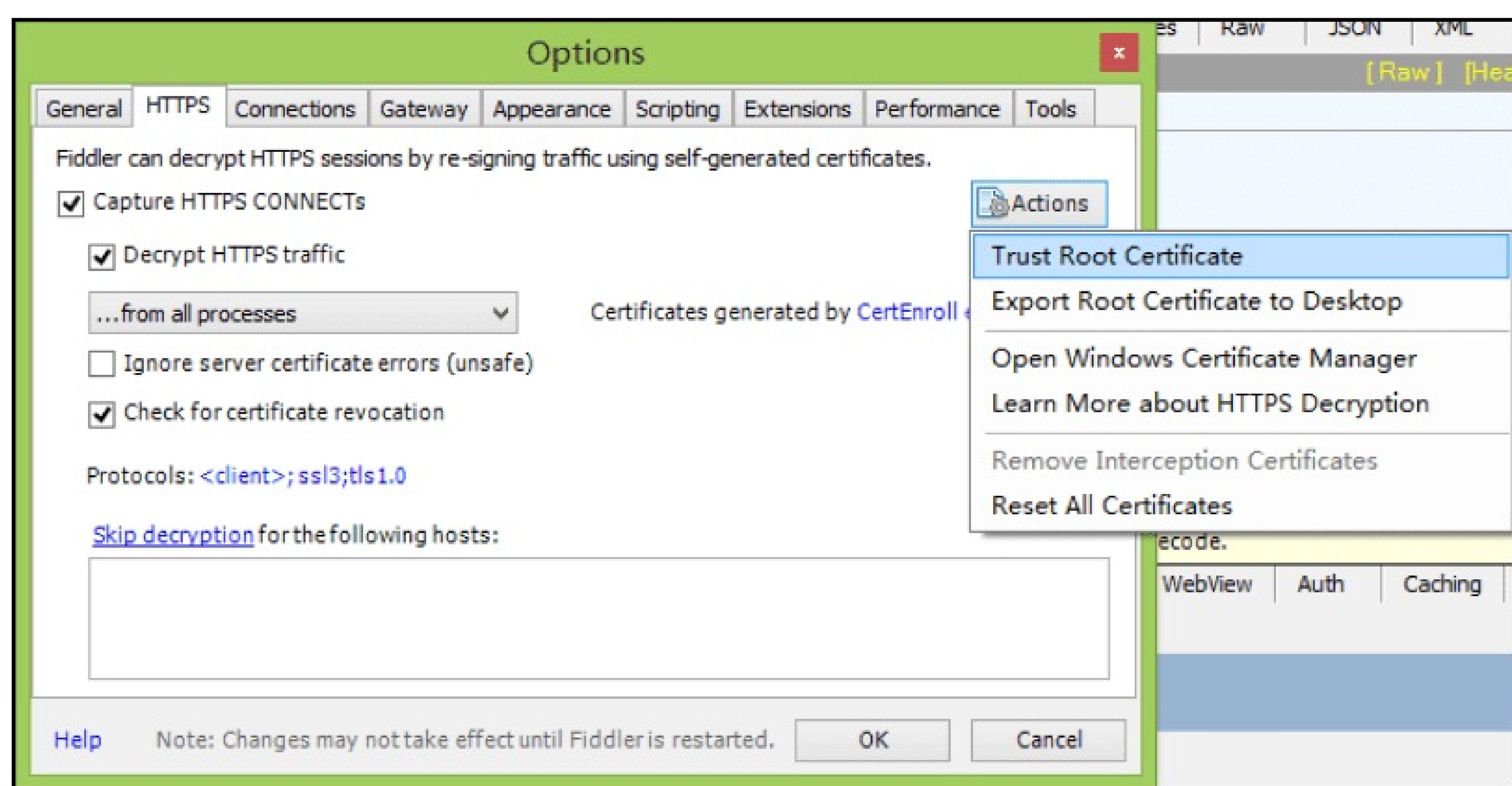


图 4-3 Fiddler 配置 HTTPS

步骤03 完成配置，重启浏览器，Fiddler 就能正常抓取 HTTPS 请求信息。

完成上述安装和配置，Fiddler 就能抓取浏览器的请求信息。除此之外，Fiddler 还能抓取手机上的请求信息，具体使用方法在后续章节会详细讲述。

4.3 Fiddler 抓取手机应用

Fiddler 可通过同一无线网络实现对手机应用的抓包，手机抓包原理和电脑抓包原理相同，手机抓包主要通过远程连接实现手机和 Fiddler 通信。

实现 Fiddler 抓取手机应用的步骤如下：

步骤01 配置 Fiddler 远程连接模式。打开 Main Menu→Tools→Fiddler Options→Connections，勾选 Allow remote computers to connect 复选框，如图 4-4 所示。

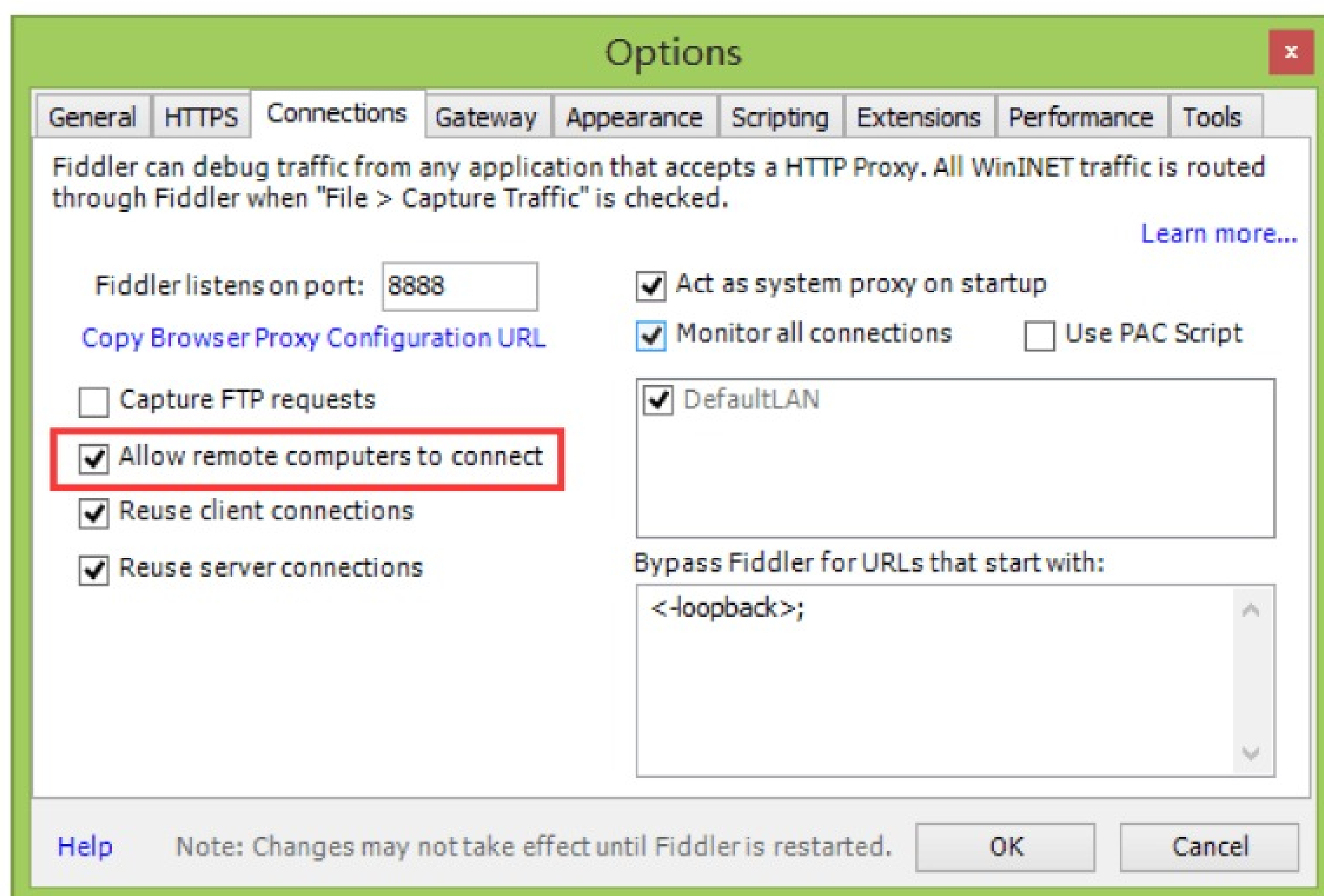


图 4-4　Fiddler 配置远程连接

步骤02 在手机端进行参数配置（以安卓手机为例）。确保手机和电脑在同一个网络，查询电脑 IP 地址，可在 CMD 下输入 ipconfig 查询（电脑 IP 为 10.168.1.240），从图 4-4 得知，Fiddler 端口为 8888（一般默认为 8888，也可自行设置）。

步骤03 在手机浏览器中输入电脑 IP 地址和 Fiddler 端口（输入“10.168.1.240: 8888”），单击确认后跳转到证书下载页面。单击下载 FiddlerRoot certificate，如图 4-5 所示。

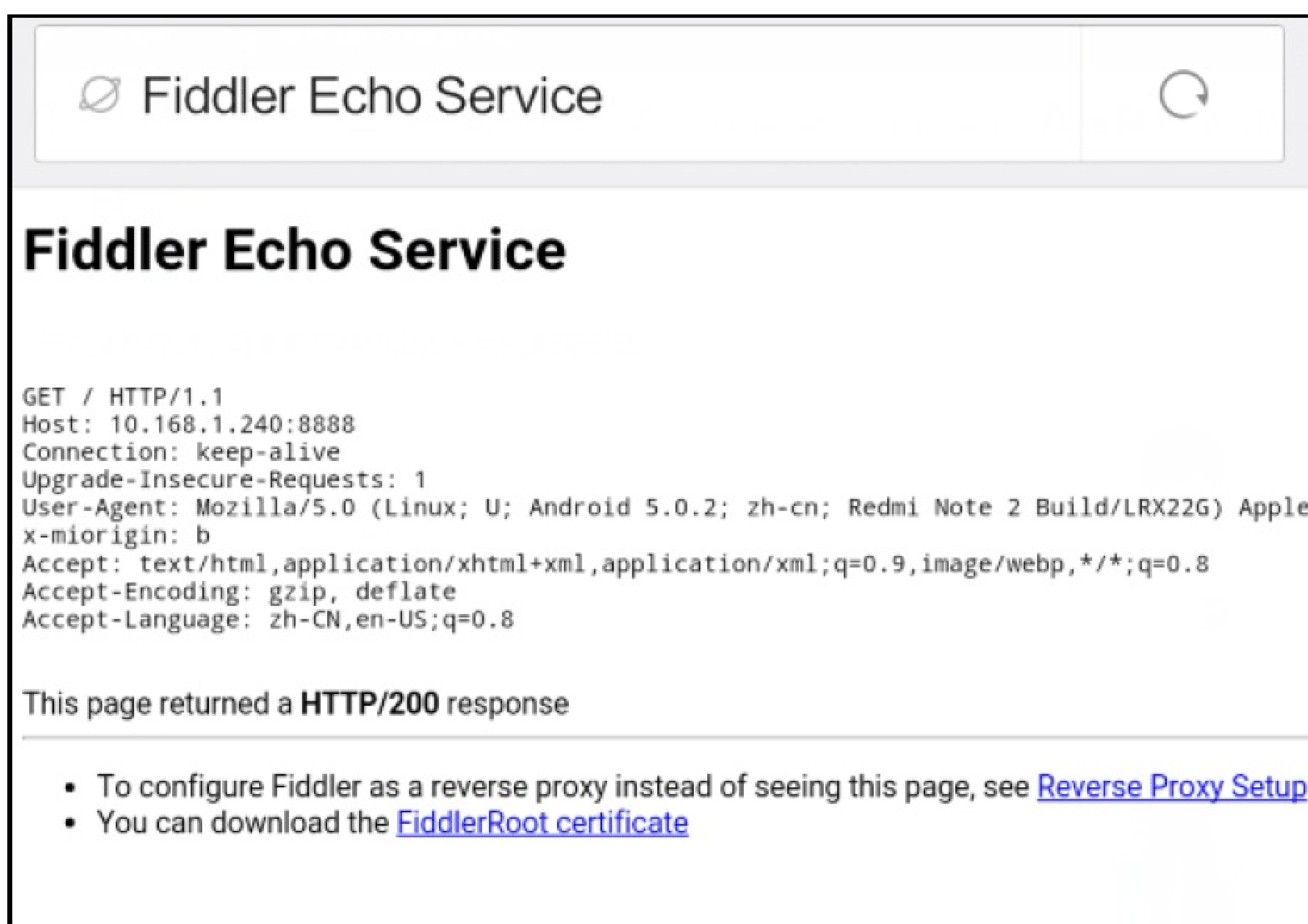


图 4-5　下载 FiddlerRoot certificate

步骤04 证书文件以 cer 为后缀名，由于不同的手机型号安装证书的方式不一致，因此这里不做详细讲述。完成证书安装后，进入手机当前连接 Wi-Fi 详情，设置代理 IP：主机名为电脑 IP 地址，端口为 Fiddler 配置的端口，如图 4-6 所示。



图 4-6　配置手机代理

步骤05 完成上述配置，可操作手机应用，在操作过程中产生的 HTTP 请求都会被电脑上的 Fiddler 抓取，如图 4-7 所示。

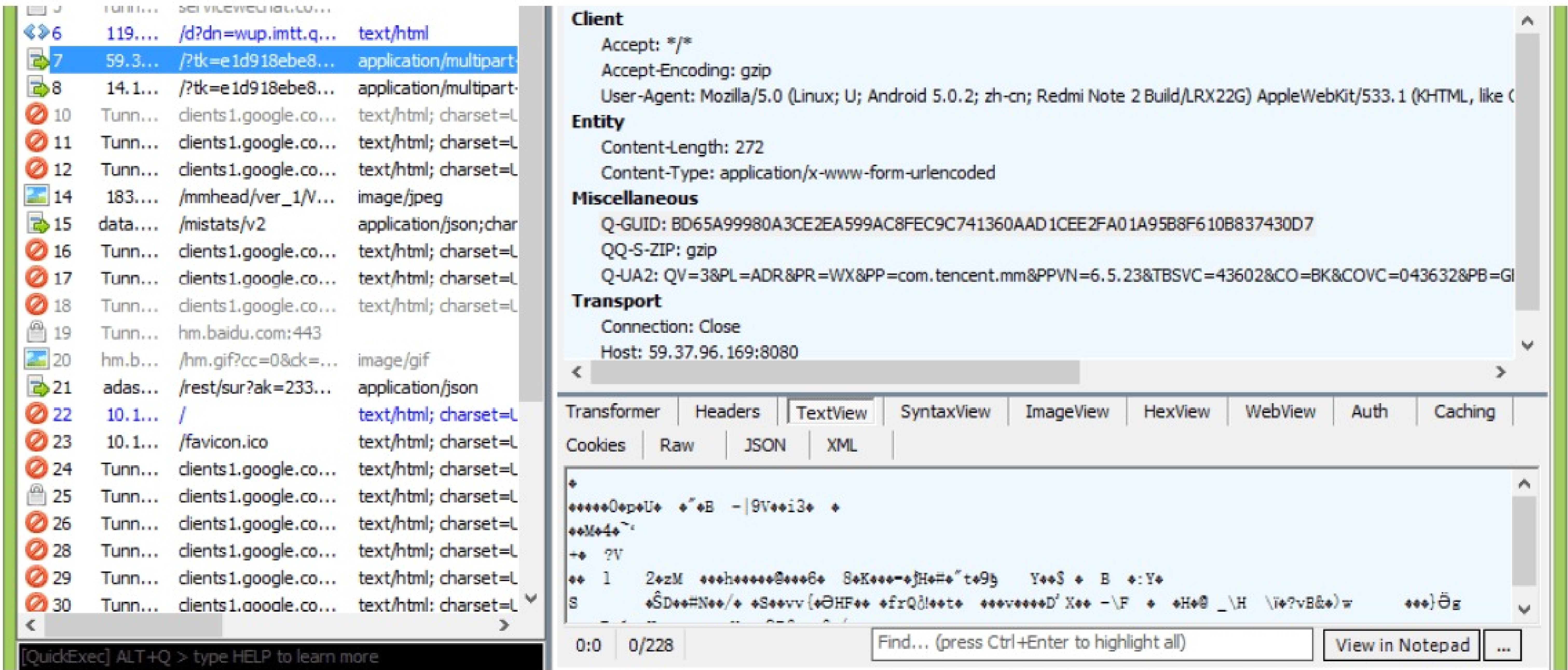


图 4-7　Fiddler 抓取手机 HTTP 请求信息

从图 4-7 中看到，User-Agent 请求头是由安卓系统发出的。同样地，若抓取 iOS 系统，也是按照上述方式配置即可。

如果停止电脑对手机的网络监控，可以回到步骤 4，将 Wi-Fi 的代理设置去掉即可。若要删除 Fiddler 证书，可在设置→系统安全→信任的凭据下找到“DO_NOT_TRUST”证书，将其删除即可。

4.4 Toolbar 工具栏

现在已可以使用 Fiddler 抓取 HTTP 请求信息了，但如何使用 Fiddler 对请求进行分析，从而获取我们所需的信息呢？要熟练掌握 Fiddler 的使用，首先要掌握其每个功能的作用。

从 4.2 节知道，Fiddler 用户界面由 6 部分组成，最为常用的是 Toolbar 工具栏、Web Session 列表、View 选项视图和 Quickexec 命令行。

Toolbar 工具栏主要提供常见的命令和设置的快捷方式，其各个按钮的功能说明如表 4-1 所示。

表 4-1 Toolbar 工具栏功能说明

快捷键	含义
	单击该按钮可以为所有选定的 Session 添加 comment
Replay	向服务器重新发送该请求
X ▾	从 Web Session 中删除已经捕捉的 Session
Go	恢复执行在 request 或 response 断点处暂停的所有 Session
Stream	打开 Stream 模式，取消所有没有设置中断的缓存
Decode	打开 Decode 模式，对请求和响应的 HTTP 内容和传输编码进行解码
Keep: All sessions ▾	选择 Web Session 列表中保存 Session 的数量
Any Process *pick target...*	单击上面的 Any Process 图标并将其移动到指定浏览器页面后，该 log 会单独记录这个页面的通信情况
Find	打开 Find Session 窗口，可快速查找某条 Session
Save	把所有的 Session 保存到 saz 文件中
(2...)	把当前桌面的屏幕截图以 JPEG 格式添加到 Web Session 列表
	简单的计时功能
Browse ▾	如果选中某个 Session，就会在 IE 中打开该 URL；如果没有选中 Session，就在 IE 中打开 about:blank
Clear Cache	清空 WinINET 的缓存文件
TextWizard	打开文本编码/解码小工具
Tearoff	新建一个包含所有 View 的窗口
MSDN Search...	在 MSDN 的 Web Session 区域进行搜索
	打开 Fiddler 的帮助窗口
	删除工具栏，如果要恢复工具栏，可单击 View→Show Toolbar

4.5 Web Session 列表

Web Session 主要以表格形式展现，需掌握表字段代表的含义、表格信息的含义以及快捷键的使用。

表字段（表头）信息的含义如表 4-2 所示。

表 4-2　Web Session 表头信息及说明

表头	含义与作用
#	对已捕捉的 Session 生成对应的 ID 号
Host	接受请求的主机名和端口
URL	请求 URL 的路径
Content-Type	Session 的内容类型
Result	响应的状态码
Protocol	网络协议类型（HTTP、HTTPS、FTP）
Body	包含的字节数
Caching	响应头中 Expires 和 Cache-Control 的值
Process	数据流在本地系统的进程
Comments	通过工具栏 Comment 按钮设置注释信息
Custom	FiddlerScript 所设置的 Ui-CustomColumn 标志位的值

观察列表每条请求信息可发现，每条数据都有不同的颜色，其颜色含义如表 4-3 所示。

表 4-3　Web Session 请求信息的颜色含义

颜色	含义
红色	表示 HTTP 状态错误
黄色	表示 HTTP 状态需用户认证
灰色	表示数据流类型 CONNECT 或表示响应内容是图像
紫色	表示响应内容是 CSS 文件
蓝色	表示响应内容是 HTML
绿色	表示响应内容是 Script 文件

除了颜色之外，每条请求信息都带有一个图标，图标含义如表 4-4 所示 。

表 4-4　Web Session 请求信息的图标含义

图标	含义
	正在向服务器发送请求
	正在向服务器获取响应
	请求停止于断点处，允许对它进行修改
	响应停止于断点处，允许对它进行修改

（续表）

图标	含义
	请求使用的是 Head 或 Options 方法，客户端无须下载内容即可获取目标 URL 和服务器信息
	请求使用 POST 方法向服务器发送数据
	响应内容是 HTML
	响应内容是图像文件
	响应内容是脚本文件
	响应内容是 CSS 文件
	响应内容是 XML 格式文件
	响应内容是 JSON 格式文件
	响应内容是音频文件
	响应内容是视频文件
	响应内容是 SilverLight 程序
	响应内容是 Flash 应用程序
	响应内容是字体文件
	响应成功
	Content-Type 内容是 Text/HTML
	响应是 HTTP / 300、301、302、303、307 重定向
	要求对客户端进行认证
	服务器返回错误的标识
	Session 被客户端或服务器中止
	响应内容使用缓存版本

通过对表字段、请求信息的颜色和请求信息的图标的了解，可知道 Fiddler 对每一种请求类型都进行了详细的划分。除此之外，用户还可以对每个请求进行操作，移动鼠标对某一条请求右击会出现操作菜单，如图 4-8 所示。

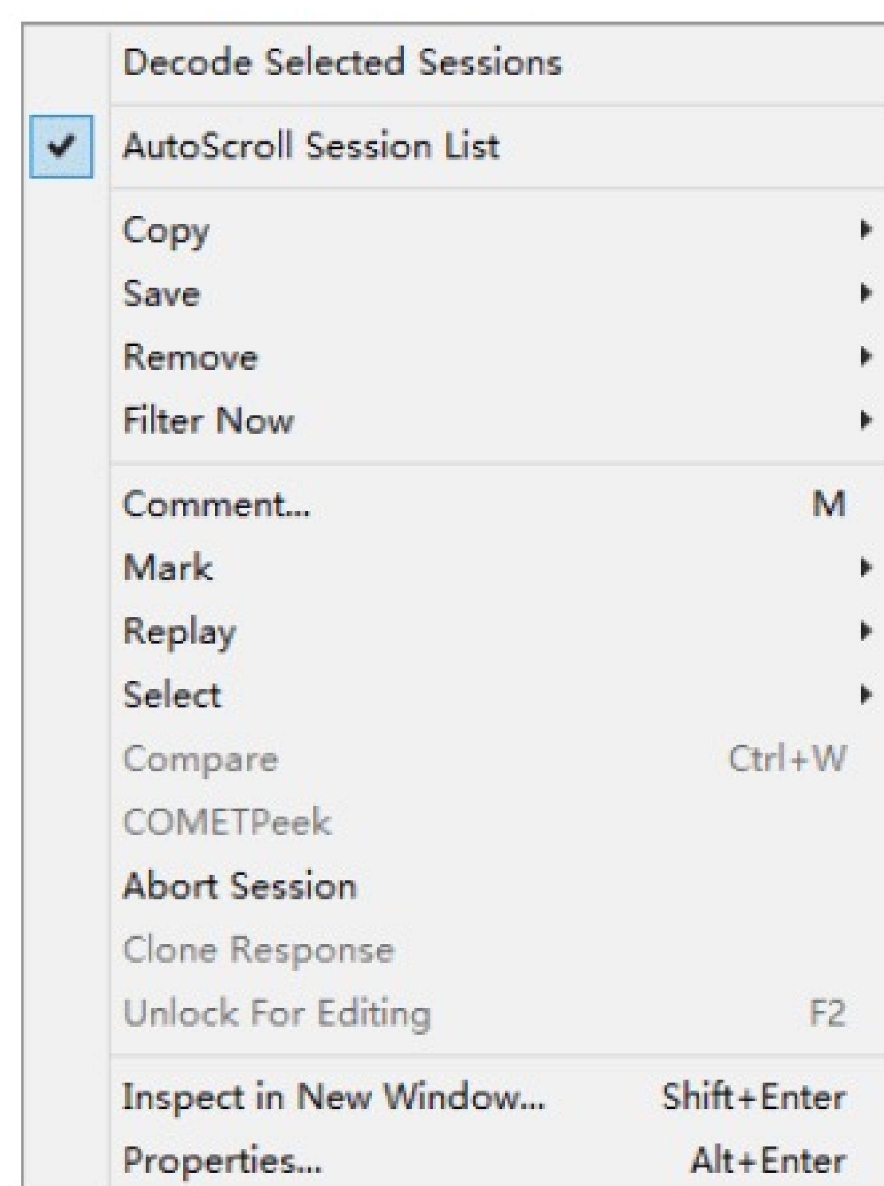


图 4-8　Web Session 操作菜单

用户可通过操作菜单对请求信息进行更改，也可以直接使用快捷键实现，快捷键如表 4-5 所示。（图中颜色较深的是常用部分。）

表 4-5 Web Session 快捷键

快捷键	含义
SpaceBar	在视图中激活并显示当前的 Session
Ctrl + A	选中所有的 Session
ESC	取消选择所有的 Session
Ctrl + I	反向选中，取消选中的 Session，选中之前未选中的 Session
Ctrl + X	删除所有 Session
Delete	删除选中的 Session
Shift + Delete	删除所有未选中的 Session
R	重新执行当前请求
Shift + R	多次执行当前的请求（在提示框输入执行次数）
U	无条件重新执行当前的请求
Shift +U	无条件多次执行当前的请求（在提示框输入执行次数）
P	选中触发该请求的父请求
C	选中该响应触发的所有子请求
D	选中与当前 Session 重复的请求
Alt + Enter	查看当前 Session 的属性
Shift + Enter	在新的 Fiddler 窗口中启动该 Session 的 Inspectors
Ctrl + 1/2/3/4/5/6	选中的 Session 分别用粗体的红色/蓝色/金色/绿色/橙色/紫色表示
M	为选中的 Session 添加描述

4.6 View 选项视图

View 主要以选项视图方式实现，将一个请求信息按功能划分在不同选项卡中，如表 4-6 所示是常用的选项视图。

表 4-6 View 常用选项视图

常用选项视图	含义
Statistics	统计选项卡，统计资源的消耗时间和数据长度等信息
Inspectors	检查选项卡，共分为两部分：请求信息和响应信息
AutoResponder	自动响应选项卡，将请求重定向到本地文件，实现人工干预 HTTP 请求
Composer	构建选项卡，用于创建 HTTP Request，并发送服务器实现模拟请求

使用 Fiddler 做爬虫开发必须掌握 Inspectors、AutoResponder 和 Composer。Inspectors 主要是对请求和响应信息进行分析和获取请求参数，如图 4-9 所示。

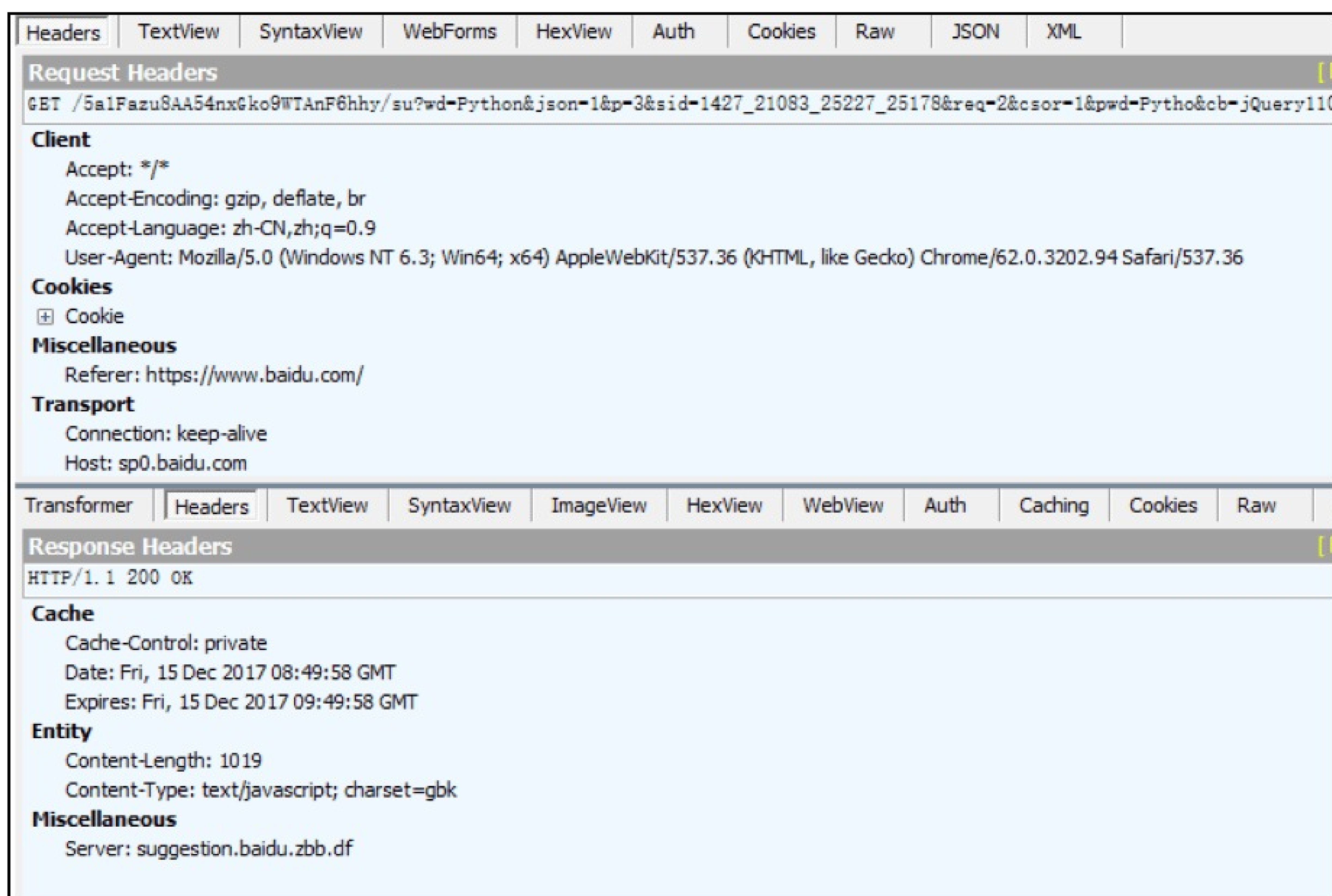


图 4-9　Inspectors 选项视图

从图 4-9 知道，Inspectors 上下划分为两个功能区。上面的功能区显示用户发送的请求信息，下面的功能区显示服务器响应的内容，每个功能区里又划分了多个选项视图。其中，Headers 选项视图显示请求头和响应头，WebForm 选项视图显示发送请求的请求参数，xxxView 选项卡显示各种类型的数据内容。

AutoResponder 和 Composer 就不多做讲解了，主要是 AutoResponder 的作用更偏向于 Web 开发调试，在爬虫开发中实用性不强；Composer 虽然能够实现对服务器的请求，但用 Fiddler 编写代码就显得本末倒置，还不如直接使用 Python 实现。

4.7　Quickexec 命令行

Quickexec 命令行可通过特定的条件快速找到符合条件的请求信息，如图 4-10 所示。

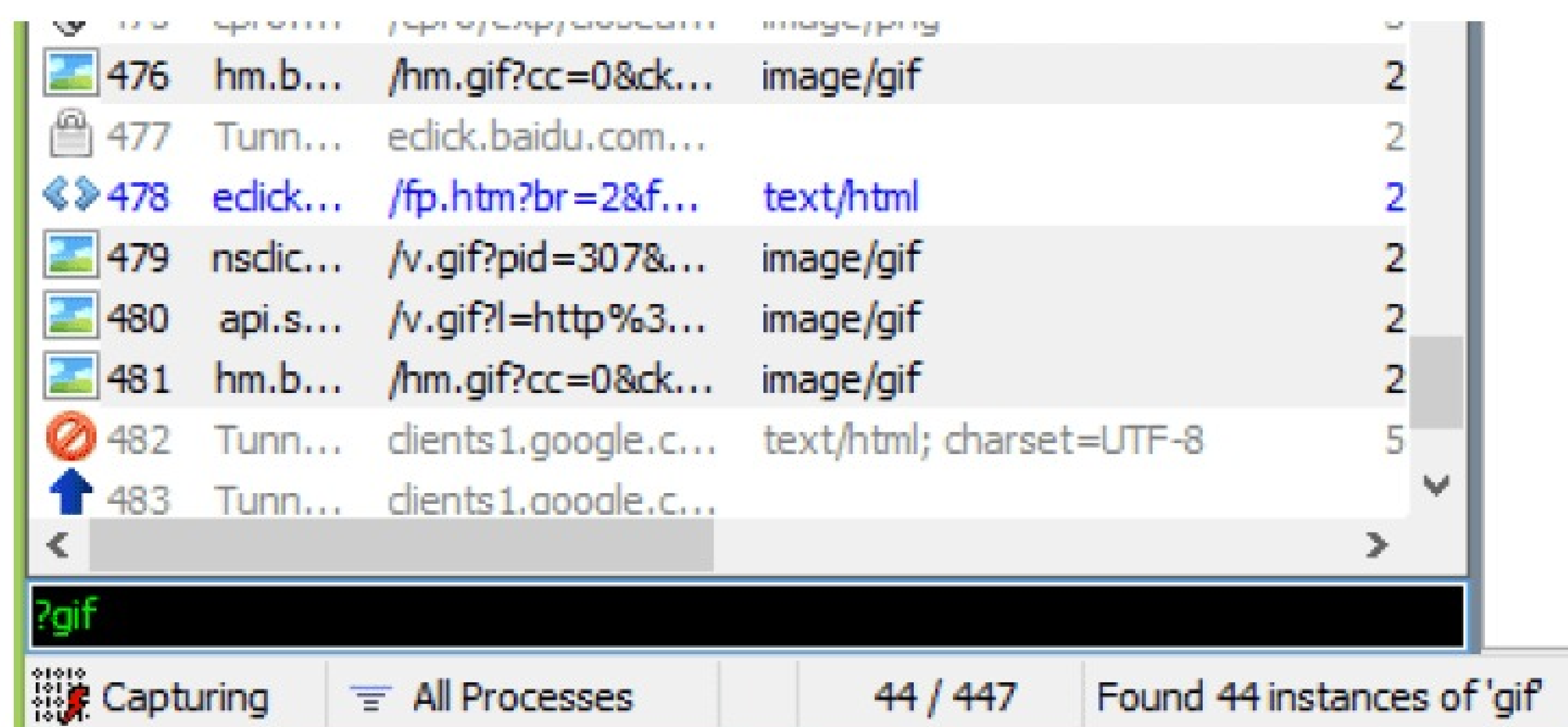


图 4-10　Quickexec 快速查找

从图 4-10 中看到，当输入“?gif”时，Web Session 列表会将符合条件的请求信息以高亮显示，在下方状态栏可以看到符合条件的请求有 44 条。

除了使用 Quickexec 实现快速查找之外，还可以使用“Ctrl+F”查找功能，如图 4-11 所示。

通过输入关键字，然后单击查找按钮，就能找到相对应的 Session 信息，而且还可以自行设定目标高亮颜色。除此之外，使用者还能根据特定的要求设置不同的查找条件。

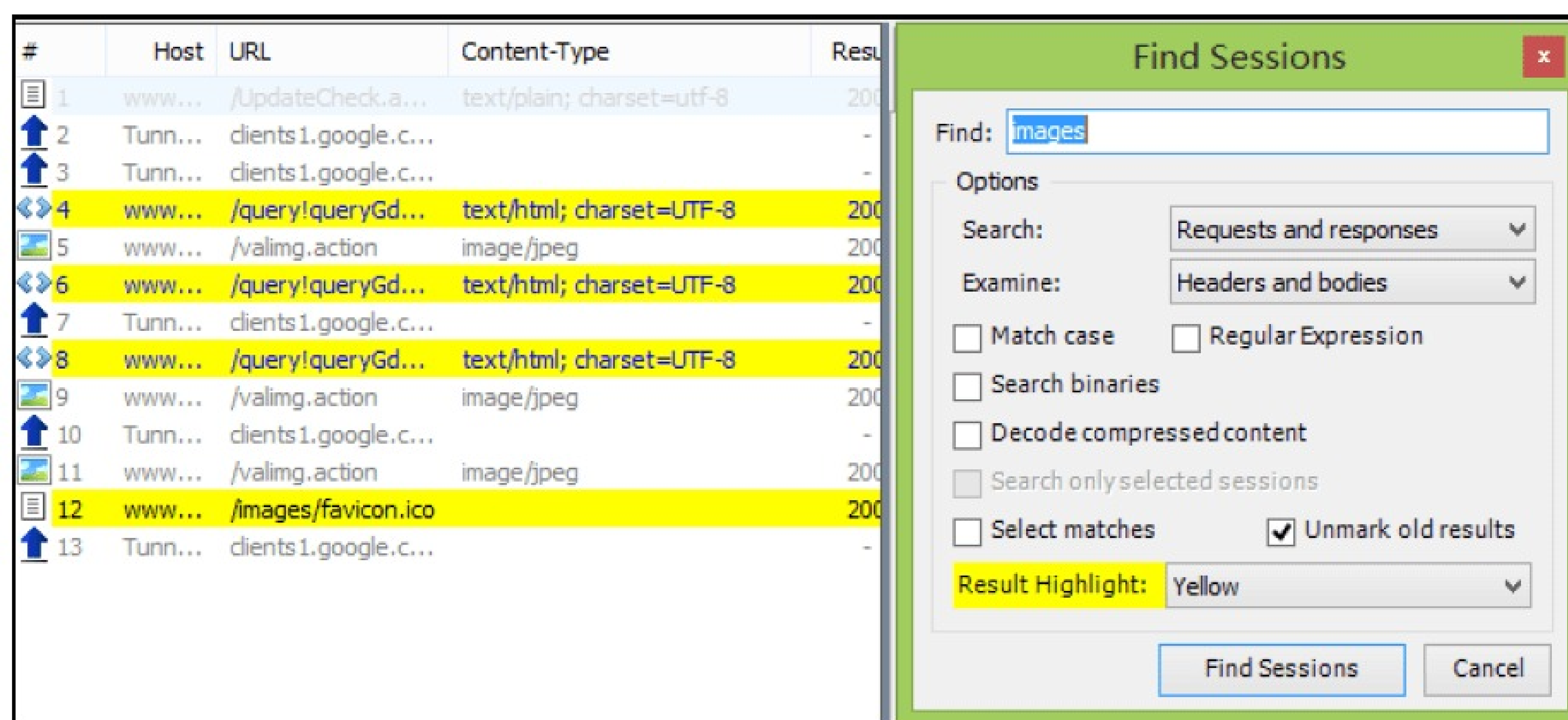


图 4-11 Ctrl+F 查找功能

4.8 本章小结

Fiddler 是一款非常流行并且实用的 HTTP 抓包工具，它的原理是在电脑中开启一个 HTTP 代理服务器，然后转发所有的 HTTP 请求和响应。因此，比一般的浏览器自带的抓包工具（开发者工具）要好用得多。不仅如此，Fiddler 还可以支持请求重放等一些高级功能，也可以支持对手机应用进行 HTTP 抓包。

Fiddler 提供了 Windows 环境下的.exe 安装包，使其安装极其简单方便。安装完成后，需配置 HTTPS 抓取功能和手机抓包功能，完成配置便可对 HTTPS 网站和手机进行抓包。

除了功能强大之外，在使用上也较为简单，使用者只要打开 Fiddler，然后在浏览器（手机）中进行操作，Fiddler 就会自动抓取请求信息。就 Fiddler 本身的功能而言，使用者只需熟知每个功能按钮的作用便知道如何使用。

对于爬虫开发人员来说，需要掌握 Web Session 列表和 View 常用选项视图的基本功能，能够分析并得知每个请求的类型、状态码、请求方式、请求头、请求链接、请求参数以及响应内容等基本信息。

第5章

爬虫库 Urllib

5.1 Urllib 简介

Urllib 是 Python 自带的标准库，无须安装，直接引用即可。Urllib 通常用于爬虫开发、API（应用程序编程接口）数据获取和测试。在 Python 2 和 Python 3 中，Urllib 在不同版本中的语法有明显的改变。

Python 2 分为 Urllib 和 Urllib2，Urllib2 可以接收一个 Request 对象，并以此来设置一个 URL 的 Headers，但是 Urllib 只接收一个 URL，意味着不能伪装用户代理字符串等。Urllib 模块可以提供进行 Urlencode 的方法，该方法用于 GET 查询字符串的生成，Urllib2 不具有这样的功能。这也是 Urllib 与 Urllib2 经常在一起使用的原因。

在 Python 3 中，Urllib 模块是一堆可以处理 URL 的组件集合，就是将 Urllib 和 Urllib2 合并在一起使用，并且命名为 Urllib。

由于 Urllib 在不同的 Python 版本上有明显的区别，在实际开发中也遇到一些尴尬的情况，其中最为主要的是版本之间的互不兼容所带来的问题。

在 Python 3 中，Urllib 是一个收集几个模块来使用 URL 的软件包，大致具备以下功能。

- urllib.request：用于打开和读取 URL。
- urllib.error：包含提出的例外 urllib.request。
- urllib.parse：用于解析 URL。
- urllib.robotparser：用于解析 robots.txt 文件。

5.2 发送请求

urllib.request.urlopen 的语法如下：

```
    urllib.request.urlopen(url, data=None, [timeout, ]*, cafile=None, capath=N
one, cadefault=False, context=None)
```

功能说明：Urllib 是用于访问 URL（请求链接）的唯一方法。

【参数解释】

- url：需要访问的网站的 URL 地址。url 格式必须完整，如 https://movie.douban.com/为完整的 url，若 url 为 movie.douban.com/，则程序运行时会提示无法识别 url 的错误。
- data：默认值为 None，Urllib 判断参数 data 是否为 None 从而区分请求方式。若参数 data 为 None，则代表请求方式为 GET；反之请求方式为 POST，发送 POST 请求。参数 data 以字典形式存储数据，并将参数 data 由字典类型转换成字节类型才能完成 POST 请求。
- timeout：超时设置，指定阻塞操作（请求时间）的超时（如果未指定，就使用全局默认超时设置）。
- cafile、capath 和 cadefault：使用参数指定一组 HTTPS 请求的可信 CA 证书。cafile 应指向包含一组 CA 证书的单个文件；capath 应指向证书文件的目录；cadefault 通常使用默认值即可。
- context：描述各种 SSL 选项的实例。

在实际使用中，常用的参数有 url、data 和 timeout。若在爬虫中遇到证书验证，则可将证书验证直接关闭，也可以设置参数指向证书的信息和位置。相比而言，设置证书比较耗时，而且通用性不强。

当对网站发送请求时，网站会返回相应的响应内容。urlopen 对象提供获取网站响应内容的方法函数，分别介绍如下。

- read()、readline()、readlines()、fileno()和 close()：对 HTTPResponse 类型数据操作。
- info()：返回 HTTPMessage 对象，表示远程服务器返回的头信息。
- getcode()：返回 HTTP 状态码。
- geturl()：返回请求的 URL。

下面的例子用于实现 Urllib 模块对网站发送请求并将响应内容写入文本文档，代码如下：

```
# 导入urllib
import urllib.request
# 打开URL
response = urllib.request.urlopen('https://movie.douban.com/', None, 2)
# 读取返回的内容
html = response.read().decode('utf8')
# 写入txt
f = open('html.txt', 'w', encoding='utf8')
f.write(html)
f.close()
```

首先导入 urllib.request 模块，然后通过 urlopen 访问一个 URL，请求方式是 GET，所以参数 data 设置为 None；最后的参数用于设置超时时间，设置为 2 秒，如果超过 2 秒，网站还没返回响应数据，就会提示请求失败的错误信息。

当得到服务器的响应后，通过 response.read()获取其响应内容。read()方法返回的是一个 bytes 类型的数据，需要通过 decode()来转换成 str 类型。最后将数据写入文本文档中，encoding 用于设置文本文档的编码格式，数据编码必须与文本文档编码一致，否则会出现乱码。运行结果如图 5-1 所示。




```
<!DOCTYPE html>
<html lang="zh-cmn-Hans" class="">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <meta name="renderer" content="webkit">
    <meta name="referrer" content="always">
    <title>
        豆瓣电影
</title>

    <meta name="baidu-site-verification" content="cZdR4xxR7RxmM4zE" />
    <meta http-equiv="Pragma" content="no-cache">
    <meta http-equiv="Expires" content="Sun, 6 Mar 2005 01:00:00 GMT">
```

图 5-1　获取豆瓣页面内容

5.3　复杂的请求

urllib.request.Request 的语法如下：

```
urllib.request.Request(url, data=None, headers={}, method=None)
```

功能说明：声明一个 request 对象，该对象可自定义 header（请求头）等请求信息。

【参数解释】

- url：完整的 url 格式，与 urllib.request.urlopen 的参数 url 一致。
- data：请求参数，与 urllib.request.urlopen 的参数 data 一致。
- headers：设置 request 请求头信息。
- method：设定请求方式，主要是 POST 和 GET 方式。

一个完整的 HTTP 请求必须要有请求头信息，而 urllib.request.Request 的作用是设置 HTTP 的请求头信息。使用 urllib.request.Request 为 5.2 节的例子设置请求头，代码如下：

```
# 导入urllib
import urllib.request
url = 'https://movie.douban.com/'
# 自定义请求头
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 6.1; WOW64)
                  AppleWebKit/537.36 (KHTML, like Gecko)
                  Chrome/45.0.2454.85 Safari/537.36
```

```
                    115Browser/6.0.3',
    'Referer': 'https://movie.douban.com/',
    'Connection': 'keep-alive'}
# 设置 request 的请求头
req = urllib.request.Request(url, headers=headers)
# 使用 urlopen 打开 req
html = urllib.request.urlopen(req).read().decode('utf-8')
# 写入文件
f = open('html.txt', 'w', encoding='utf8')
f.write(html)
f.close()
```

5.4 代理 IP

代理 IP 的原理：以本机先访问代理 IP，再通过代理 IP 地址访问互联网，这样网站（服务器）接收到的访问 IP 就是代理 IP 地址。

Urllib 提供了 urllib.request.ProxyHandler()方法可动态设置代理 IP 池，代理 IP 主要以字典格式写入方法。完成代理 IP 设置后，将设置好的代理 IP 写入 urllib.request.build_opener()方法，生成对象 opener，然后通过 opener 的 open()方法向网站（服务器）发送请求。

沿用前面章节的例子，将例子改为使用代理 IP 访问网站，代码如下：

```
import urllib.request
url = 'https://movie.douban.com/'
# 设置代理 IP
proxy_handler = urllib.request.ProxyHandler({
    'http': '218.56.132.157:8080',
    'https': '183.30.197.29:9797'})
# 必须使用 build_opener()函数来创建带有代理 IP 功能的 opener 对象
opener = urllib.request.build_opener(proxy_handler)
response = opener.open(url)
html = response.read().decode('utf-8')
f = open('html.txt', 'w', encoding='utf8')
f.write(html)
f.close()
```

注意，由于使用代理 IP，因此连接 IP 的时候有可能出现超时而导致报错，遇到这种情况只要更换其他代理 IP 地址或者再次访问即可。以下是常见的报错信息。

- ConnectionResetError: [WinError 10054]远程主机强迫关闭了一个现有的连接。
- urllib.error.URLError: urlopen error Remote end closed connection without response（结束没有响应的远程连接）。
- urllib.error.URLError: urlopen error [WinError 10054] 远程主机强迫关闭了一个现有的连接。
- TimeoutError: [WinError 10060] 由于连接方在一段时间后没有正确答复或连接的主机没有反应，因此连接尝试失败。
- urllib.error.URLError: urlopen error [WinError 10061] 由于目标计算机拒绝访问，因此无法连接。

5.5　使用 Cookies

Cookies 主要用于获取用户登录信息，比如，通过提交数据实现用户登录之后，会生成带有登录状态的 Cookies，这时可以将 Cookies 保存在本地文件中，下次程序运行的时候，可以直接读取 Cookies 文件来实现用户登录。特别对于一些复杂的登录，如验证码、手机短信验证登录这类网站，使用 Cookies 能简单解决重复登录的问题。

Urllib 提供 HTTPCookieProcessor()对 Cookies 操作，但 Cookies 的读写是由 MozillaCookieJar()完成的。下面的例子实现 Cookies 写入文件，代码如下：

```
import urllib.request
from http import cookiejar
filename = 'cookie.txt'
# MozillaCookieJar 保存 cookie
cookie = cookiejar.MozillaCookieJar(filename)
# HTTPCookieProcessor 创建 cookie 处理器
handler = urllib.request.HTTPCookieProcessor(cookie)
# 创建自定义 opener
opener = urllib.request.build opener(handler)
# open 方法打开网页
response = opener.open('https://movie.douban.com/')
# 保存 cookie 文件
cookie.save()
```

代码中的 cookiejar 是自动处理 HTTP Cookie 的类，MozillaCookieJar()用于将 Cookies 内容写入文件。程序运行时先创建 MozillaCookieJar()对象，然后将对象直接传入函数 HTTPCookieProcessor()，生成 opener 对象；最后使用 opener 对象访问 URL，访问过程所生成的 Cookies 就直接写入已创建的文本文档中。

接着再看如何读取 Cookies，代码如下：

```
import urllib.request
from http import cookiejar
filename = 'cookie.txt'
# 创建 MozillaCookieJar 对象
cookie = cookiejar.MozillaCookieJar()
# 读取 cookie 内容到变量
cookie.load(filename)
# HTTPCookieProcessor 创建 cookie 处理器
handler = urllib.request.HTTPCookieProcessor(cookie)
# 创建 opener
opener = urllib.request.build_opener(handler)
# opener 打开网页
response = opener.open('https://movie.douban.com/')
# 输出结果
print(cookie)
```

读取和写入的方法很相似，主要区别在于：两者对 MozillaCookieJar()对象的操作不同，导致实现功能也不同。运行结果如图 5-2 所示。

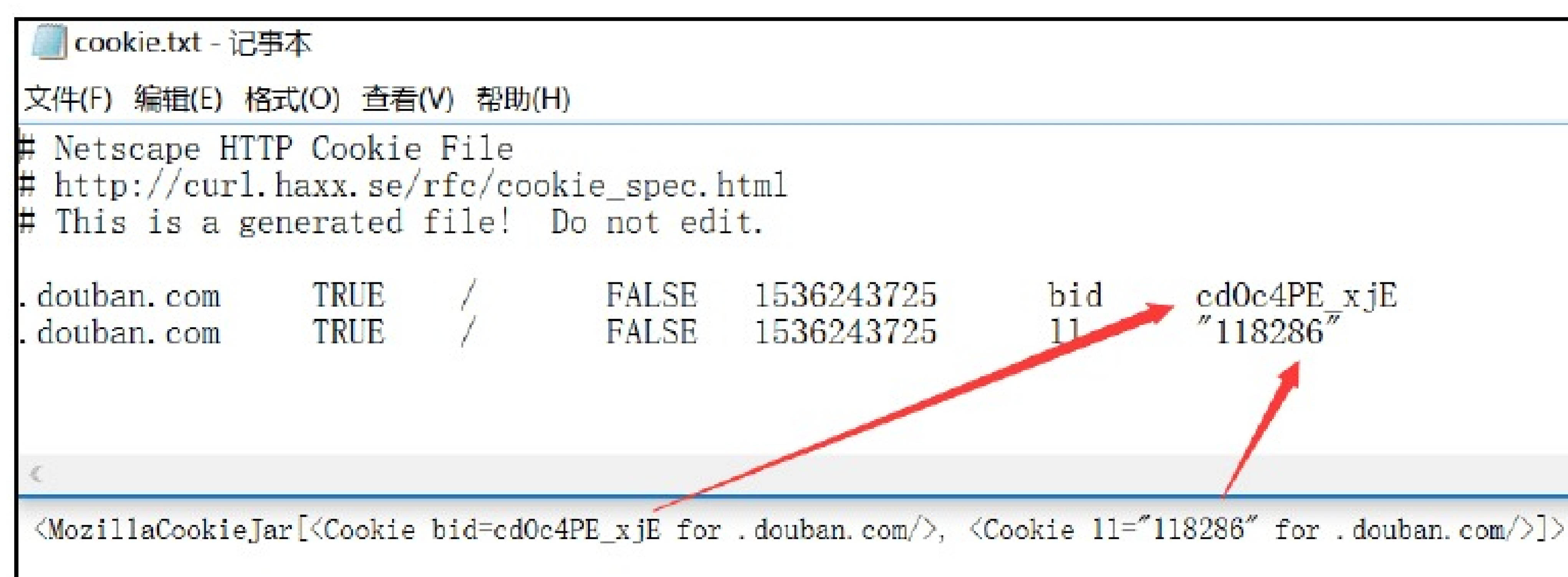


图 5-2 验证 Cookies

注意，为了方便测试，上述代码中使用的 cookie.save()和 cookie.load(filename)将 Cookies 内容显示在文本文档中。在实际开发中，为了提高安全性，可以在保存和读取 Cookies 时设置参数，使 Cookies 信息隐藏在文件中。方法如下：

```
cookie.save(ignore_discard=True,ignore_expires=True)
cookie.load(filename, ignore_discard=True, ignore_expires=True)
```

5.6 证书验证

当遇到一些特殊的网站时，在浏览器上会显示连接不是私密连接而导致无法浏览该网页。若在没有安装 12306 根证书的情况下访问 12306 网站，则页面如图 5-3 所示。



图 5-3 查询 12306 的车票

这里补充一个知识点，CA 证书也叫 SSL 证书，是数字证书的一种，类似于驾驶证、护照和营业执照的电子副本。因为配置在服务器上，也称为 SSL 服务器证书。

SSL 证书就是遵守 SSL 协议，由受信任的数字证书机构颁发 CA，在验证服务器身份后颁发，具有服务器身份验证和数据传输加密功能。

SSL 证书在客户端浏览器和 Web 服务器之间建立一条 SSL 安全通道（Secure Socket Layer，SSL），安全协议是由 Netscape Communication 公司设计开发的。该安全协议主要用来提供对用户和服务器的认证，对传送的数据进行加密和隐藏，确保数据在传送中不被改变，即数据的完整性，现已成为该领域中全球化的标准。

一些特殊的网站会使用自己的证书，如 12306 首页提示下载安装根证书，这是为了确保网站

的数据在传输过程中的安全性。在讲述 urllib.request.urlopen 的时候，urlopen 带有 cafile、capath 和 cadefault 参数，可以用于设置用户的 CA 证书。

遇到这类验证证书的网站，最简单而暴力的方法是直接关闭证书验证，可以在代码中引入 SSL 模块，设置关闭证书验证即可。代码如下：

```
import urllib.request
import ssl
# 关闭证书验证
ssl. create default https context=ssl. create unverified context
url = 'https://kyfw.12306.cn/otn/leftTicket/init'
response = urllib.request.urlopen(url)
# 输出状态码
print(response.getcode())
```

5.7　数 据 处 理

我们知道 urllib.request.urlopen()方法是不区分请求方式的，识别请求方式主要通过参数 data 是否为 None。如果向服务器发送 POST 请求，那么参数 data 需要使用 urllib.parse 对参数内容进行处理。

Urllib 在请求访问服务器的时候，如果发生数据传递，就需要对内容进行编码处理，将包含 str 或 bytes 对象的两个元素元组序列转换为百分比编码的 ASCII 文本字符串。如果字符串要用作 POST，那么它应该被编码为字节，否则会导致 TypeError 错误。

Urllib 发送 POST 请求的方法如下：

```
import urllib.request
import urllib.parse
url = 'https://movie.douban.com/'
data = {
    'value': 'true',
}
# 数据处理
data = urllib.parse.urlencode(data).encode('utf-8')
req = urllib.request.urlopen(url, data=data)
```

代码中 urllib.parse.urlencode(data)将数据转换成字节的数据类型，而 encode('utf-8')设置字节的编码格式。这里需要注意的是，编码格式主要根据网站的编码格式来决定。urlencode()的作用只是对请求参数做数据格式转换处理。

除此之外，Urllib 还提供 quote()和 unquote()对 URL 编码处理，使用方法如下：

```
import urllib.parse
url = '%2523%25E7%25BC%2596%25E7%25A8%258B%2523'
# 第一次解码
first = urllib.parse.unquote(url)
print(first)
# 输出：'%23%E7%BC%96%E7%A8%8B%23'
# 第二次解码
second = urllib.parse.unquote(first)
```

```
print(second)
# 输出：'#编程#'
```

上述例子将已编码处理的 URL 进行解码还原，同样的方法，可使用 quote()对数据进行编码处理。quote()和 unquote()的作用是解决请求参数中含有中文内容的问题。

5.8 本章小结

本章主要讲解了 Python 自带模块 Urllib 的功能和使用。Urllib 通常用于爬虫开发和 API（应用程序编程接口）数据获取和测试。在 Python 2 和 Python 3 中，Urllib 的语法有明显的改变。其常用的语法有以下几种。

- urllib.request.urlopen：urllib 最基本的使用功能，用于访问 URL（请求链接）的唯一方法。
- urllib.request.Request：声明 request 对象，该对象可自定义请求头（header）、请求方式等信息。
- urllib.request.ProxyHandler：动态设置代理 IP 池，可加载请求对象。
- urllib.request.HTTPCookieProcessor：设置 Cookies 对象，可加载请求对象。
- urllib.request.build_opener()：创建请求对象，用于代理 IP 和 Cookies 对象加载。
- urllib.parse.urlencode(data).encode('utf-8')：请求数据格式转换。
- urllib.parse.quote(url)：URL 编码处理，主要对 URL 上的中文等特殊符号编码处理。
- urllib.parse.unquote(url)：URL 解码处理，将 URL 上的特殊符号还原。

除了 Urllib 之外，一些特殊请求需要结合其他模块配合使用，如 Cookies 读写由 HTTP 模块完成，关闭证书验证需要 SSL 模块设置，等等。

第 6 章

爬虫库 Requests

6.1　Requests 简介及安装

Requests 是 Python 的一个很实用的 HTTP 客户端库，完全满足如今网络爬虫的需求。与 Urllib 对比，Requests 不仅具备 Urllib 的全部功能；在开发使用上，语法简单易懂，完全符合 Python 优雅、简洁的特性；在兼容性上，完全兼容 Python 2 和 Python 3，具有较强的适用性。

Requests 可通过 pip 安装，具体如下。

- Windows 系统：pip install requests。
- Linux 系统：sudo pip install requests。

除了使用 pip 安装之外，还可以下载 whl 文件安装，方法如下：

（1）访问 www.lfd.uci.edu/~gohlke/pythonlibs，按 Ctrl+F 组合键搜索关键字“requests”，如图 6-1 所示。



图 6-1　安装 requests

（2）单击下载 requests□2.20.0□py2.py3□none□any.whl，把下载文件直接解压，将解压出来的文件直接放入 Python 的安装目录 Lib\site-packages 中即可。

（3）除了解压 whl，还可以使用 pip 安装 whl 文件。例如把下载的文件保存在 E 盘，打开 CMD

（终端），将路径切换到 E 盘，输入安装命令：

```
E:\>pip install requests-2.20.0-py2.py3-none-any.whl
```

完成 Requests 安装后，在终端（CMD）下运行 Python，查看 Requests 版本信息，检测是否安装成功。方法如下：

```
E:\>python
>>> import requests
>>> requests.  version
'2.20.0'
```

6.2 请 求 方 式

HTTP 的常用请求是 GET 和 POST，Requests 对此区分两种不同的请求方式。GET 请求有两种形式，分别是不带参数和带参数，以百度为例：

```
# 不带参数
https://www.baidu.com/
# 带参数 wd
https://www.baidu.com/s?wd=python
```

判断 URL 是否带有参数，可以对符号“?”判断。一般网址末端（域名）带有“?”，就说明该 URL 是带有请求参数的，反之则不带有参数。GET 参数说明如下：

（1）wd 是参数名，参数名由网站（服务器）规定。

（2）python 是参数值，可由用户自行设置。

（3）如果一个 URL 有多个参数，参数之间用“&”连接。

Requests 实现 GET 请求，对于带参数的 URL 有两种请求方式：

```
import requests
# 第一种方式
r = requests.get('https://www.baidu.com/s?wd=python')
# 第二种方式
url = 'https://www.baidu.com/s'
params = {'wd': 'python'}
# 左边 params 在 GET 请求中表示设置参数
r = requests.get(url, params=params)
# 输出生成的 URL
print(r.url)
```

两种方式都是请求同一个 URL，在实际开发中建议使用第一种方式，因为代码简洁，如果参数是动态变化的，那么可使用字符串格式化对 URL 动态设置，例如 'https://www.baidu.com/s?wd=%s' %('python')。

POST 请求是我们常说的提交表单，表单的数据内容就是 POST 的请求参数。Requests 实现 POST 请求需设置请求参数 data，数据格式可以为字典、元组、列表和 JSON 格式，不同的数据格式有不同的优势。代码如下：

```
# 字典类型
data = {'key1': 'value1', 'key2': 'value2'}
# 元组或列表
(('key1', 'value1'), ('key1', 'value2'))
# JSON
import json
data = {'key1': 'value1', 'key2': 'value2'}
# 将字典转换 JSON
data=json.dumps(data)
# 发送 POST 请求
r = requests.post("https://www.baidu.com/", data=data)
print(r.text)
```

可以看出，左边的 data 是 POST 方法的参数，右边的 data 是发送请求到网站（服务器）的数据。值得注意的是，Requests 的 GET 和 POST 方法的请求参数分别是 params 和 data，别混淆两者的使用要求。

当向网站（服务器）发送请求时，网站会返回相应的响应（response）对象，包含服务器响应的信息。Requests 提供以下方法获取响应内容。

- r.status_code：响应状态码。
- r.raw：原始响应体，使用 r.raw.read() 读取。
- r.content：字节方式的响应体，需要进行解码。
- r.text：字符串方式的响应体，会自动根据响应头部的字符编码进行解码。
- r.headers：以字典对象存储服务器响应头，但是这个字典比较特殊，字典键不区分大小写，若键不存在，则返回 None。
- r.json()：Requests 中内置的 JSON 解码器。
- r.raise_for_status()：请求失败（非 200 响应），抛出异常。
- r.url：获取请求链接。
- r.cookies：获取请求后的 cookies。
- r.encoding：获取编码格式。

6.3　复杂的请求方式

从第 5 章得知，复杂的请求方式通常有请求头、代理 IP、证书验证和 Cookies 等功能。Requests 将这一系列复杂的请求做了简化，将这些功能在发送请求中以参数的形式传递并作用到请求中。

（1）添加请求头：请求头以字典的形式生成，然后发送请求中设置的 headers 参数，指向已定义的请求头，代码如下：

```
headers = {
    'content-type': 'application/json',
    'User-Agent': 'Mozilla/5.0 (Windows NT 6.3; WOW64;
                rv:41.0) Gecko/20100101 Firefox/41.0'}
requests.get("https://www.baidu.com/", headers=headers)
```

（2）使用代理 IP：代理 IP 的使用方法与请求头的使用方法一致，设置 proxies 参数即可，代码如下：

```
import requests
proxies = {
  "http": "http://10.10.1.10:3128",
  "https": "http://10.10.1.10:1080",
}
requests.get("https://www.baidu.com/", proxies=proxies)
```

（3）证书验证：通常设置关闭验证即可。在请求设置参数 verify=False 时就能关闭证书的验证，默认情况下是 True。如果需要设置证书文件，那么可以设置参数 verify 值为证书路径。

```
import requests
url = 'https://kyfw.12306.cn/otn/leftTicket/init'
# 关闭证书验证
r = requests.get(url, verify=False)
print(r.status code)
# 开启证书验证
# r = requests.get(url, verify=True)
# 设置证书所在路径
# r = requests.get(url, verify= '/path/to/certfile')
```

（4）超时设置：发送请求后，由于网络、服务器等因素，请求到获得响应会有一个时间差。如果不想程序等待时间过长或者延长等待时间，可以设定 timeout 的等待秒数，超过这个时间之后停止等待响应。如果服务器在 timeout 秒内没有应答，将会引发一个异常。使用代码如下：

```
requests.get("https://www.baidu.com/", timeout=0.001)
requests.post("https://www.baidu.com/", timeout=0.001)
```

（5）使用 Cookies：在请求过程中使用 Cookies 也只需设置参数 Cookies 即可。Cookies 的作用是标识用户身份，在 Requests 中以字典或 RequestsCookieJar 对象作为参数。获取方式主要是从浏览器读取和程序运行所产生。下面的例子进一步讲解如何使用 Cookies，代码如下：

```
import requests
temp cookies='JSESSIONID GDS=y4p7osFr IYV5Udyd6c1drWE8MeTpQn0Y58Tg8cCONVP0
20y2N!450649273;name=value'
cookies dict = {}
for i in temp cookies.split(';'):
    value = i.split('=')
    cookies dict [value[0]] = value[1]
r = requests.get(url, cookies=cookies)
print(r.text)
```

代码中变量 temp_cookies 是 Cookies 信息，可以在 Chrome 开发者工具→Network →某请求的 Headers→Request Headers 中找到 Cookie 所对应的值。然后将字符串转换成字典格式，转换规则主要执行两次分割：第一次以"；"分割，得到列表 A，第二次是列表 A 的每一个元素以"="分割，得到字典的键值对。

当程序发送请求时（不设参数 cookies），会自动生成一个 RequestsCookieJar 对象，该对象用于存放 Cookies 信息。Requests 提供 RequestsCookieJar 对象和字典对象的相互转换，代码如下：

```
import requests
```

```
url = 'https://movie.douban.com/'
r = requests.get(url)
# r.cookies 是 RequestsCookieJar 对象
print(r.cookies)
mycookies = r.cookies

# RequestsCookieJar 转换字典
cookies dict = requests.utils.dict from cookiejar(mycookies)
print(cookies dict)

# 字典转换 RequestsCookieJar
cookies jar = requests.utils.cookiejar from dict(cookies dict,
              cookiejar=None, overwrite=True)
print(cookies jar)

# 在 RequestsCookieJar 对象添加 Cookies 字典中
print(requests.utils.add_dict_to_cookiejar(mycookies, cookies_dict))
```

如果要将 Cookies 写入文件，可使用 http 模块实现 Cookies 的读写。除此之外，还可以将 Cookies 以字典形式写入文件，此方法相比 http 模块读写 Cookies 更为简单，但安全性相对较低。使用方法如下：

```
import requests
url = 'https://movie.douban.com/'
r = requests.get(url)
# RequestsCookieJar 转换字典
cookies dict = requests.utils.dict from cookiejar(mycookies)
# 写入文件
f = open('cookies.txt', 'w', encoding='utf-8')
f.write(str(cookies dict))
f.close()
# 读取文件
f = open('cookies.txt', 'r')
dict value = f.read()
f.close()
# eval(dict value)将字符串转换为字典
print(eval(dict value))
r = requests.get(url, cookies=eval(dict value))
print(r.status_code)
```

6.4　下载与上传

下载文件主要从服务器获取文件内容，然后将内容保存到本地。下载文件的方法如下：

```
import requests
url = 'https://www.python.org/static/img/python-logo.png'
r = requests.get(url)
f = open('python.jpg', 'wb')
# r.content 获取响应内容（字节流）
f.write(r.content)
f.close()
```

代码变量 url 是一个图片文件 URL 地址，对文件所在 URL 地址发送请求（大多数是 GET 请求方式）；服务器将文件内容作为响应内容，然后将得到的内容以字节流（Bytes）格式写入自定义文件，这样就能实现文件下载。

除了文件下载外，还有更为复杂的文件上传，文件上传是将本地文件以字节流的方式上传到服务器，再由服务器接收上传内容，并做出相应的响应。文件上传存在一定的难度，其难点在于服务器接收规则不同，不同的网站接收的数据格式和数据内容会不一致。下面以发送图片微博为例进行介绍。

（1）在浏览器中输入 https://weibo.cn/，在网页上单击“高级”按钮并使用 Fiddler 抓包工具（由于发送微博时，网页发生 302 跳转，因此使用 Chrome 会清空请求信息，导致抓取难度较大）。

（2）单击“选择文件”，选择图片文件并输入发布内容“Python 爬虫”，最后单击“发布”按钮发布微博。查看 Fiddler 抓取的请求信息，如图 6-2 所示。

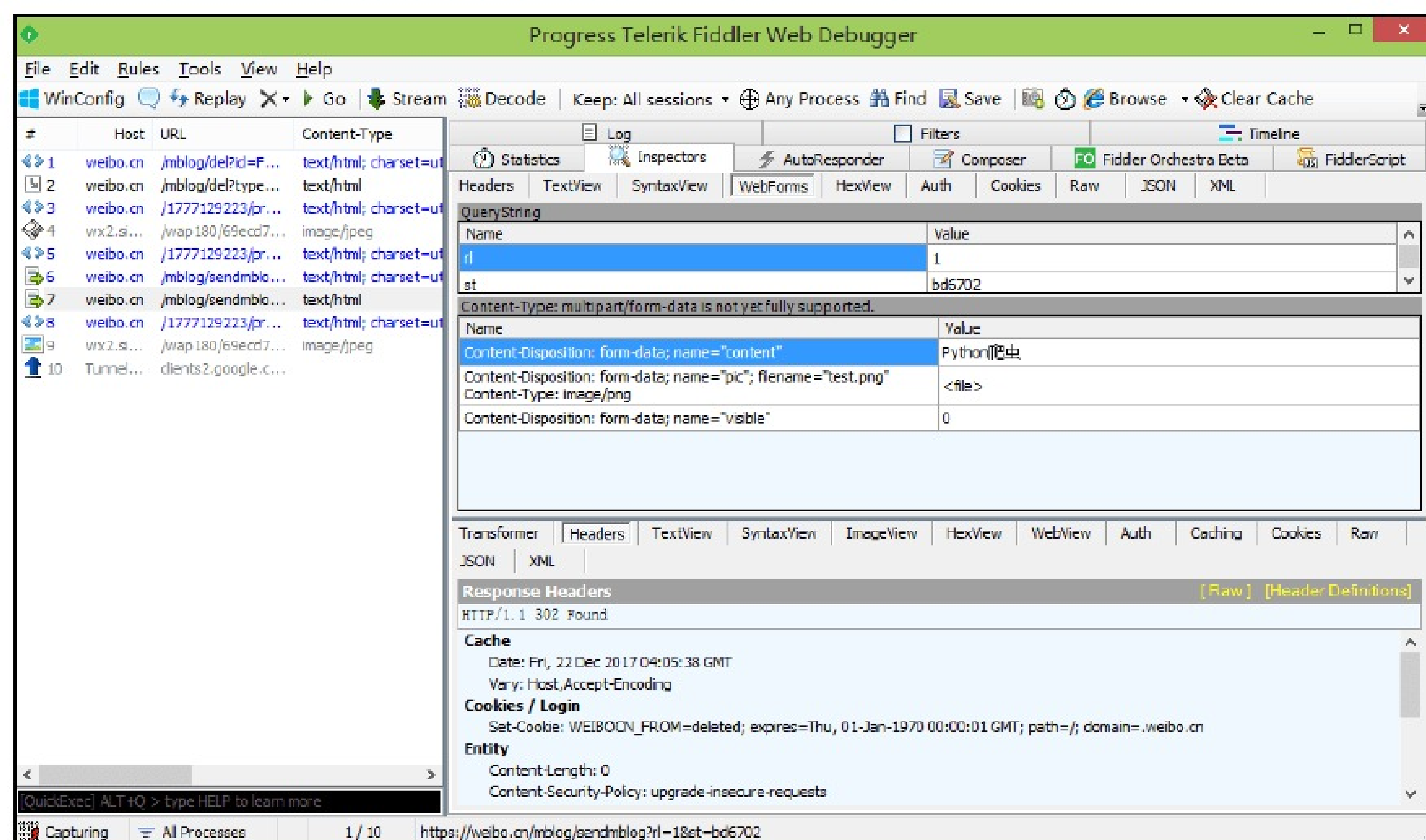


图 6-2　Fiddler 抓取的请求信息

从图 6-2 得知，该请求方式是 POST，QueryString 是 POST 的请求参数 data，Content-type 是上传文件，三个 Content-Disposition 分别对应发布内容、发布图片和设置分组可见。代码实现如下：

```
url = 'https://weibo.cn/mblog/sendmblog?rl=0&st=bd6702'
cookies = {'xxx': 'xxx'}
files = {'content': (None, 'Python 爬虫'),
       'pic': ('pic', open('test.png', 'rb'),
       'image/png'),'visible': (None, '0')}
r = requests.post(url, files=files, cookies=cookies)
print(r.status_code)
```

POST 数据对象是以文件为主的，上传文件时使用 files 参数作为请求参数。Requests 对提交的数据和文件所使用的请求参数做了明确的规定。

参数 files 也是以字典形式传递的，每个 Content-Disposition 为字典的键值对，Content-Disposition 的 name 为字典的键，value 为字典的值。

此外，不同的网站设置对 files 参数的设置也是不一样的，下面列出较为常见的上传方法：

```
#单独一个文件请求
{
  "field1" : open("filePath1", "rb").read())
}

#同时选中多个文件
{
  "field1" : [
        ("filename1", open("filePath1", "rb")),
        ("filename2", open("filePath2", "rb"), "image/png"),
        open("filePath3", "rb"),
        open("filePath4", "rb").read()
        ]
}
```

6.5　本章小结

Requests 是 Python 的一个很实用的 HTTP 客户端库，可完全满足如今编写网络爬虫程序的需求，是爬虫开发人员首选的爬虫库。其具有语法简单易懂，完全符合 Python 优雅和简洁的特性，在兼容性上完全兼容 Python 任何版本，具有较强的适用性。

读者要掌握 Requests 实现 GET 和 POST 请求时分别使用了不同的方法，如下代码所示：

```
import requests
url = 'https://baidu.com/'
# GET 请求
r = requests.get(url,headers=headers,
     proxies=proxies,verify=False,cookies=cookies)
# POST 请求
r = requests.post(url,data=data,files=files,
     headers=headers, proxies=proxies,
     verify=False, cookies=cookies)
```

Requests 的 GET 和 POST 将请求中所需要使用的功能都以参数的形式直接作用到请求中。一个发送请求的语句就已包含了请求头、代理 IP、Cookies、证书验证、文件上传等功能。

另外，Requests 还提供了 r.status_code、r.raw、r.content、r.text、r.headers、r.json()、r.raise_for_status()、r.url、r.cookies、r.encoding 10 种方法获取响应内容。

第 7 章

Requests-Cache 爬虫缓存

7.1 简介及安装

Requests-Cache 是 Requests 模块的一个扩展功能，它是根据 Requests 的发送请求来生成相应的缓存数据。当 Requests 重复向同一个 URL 发送请求的时候，Requests-Cache 会判断当前请求是否已产生缓存，若已有缓存，则从缓存里读取数据作为响应内容；若没有缓存，则向网站服务器发送请求，并将得到的响应内容写入相应的数据库里。

Requests-Cache 的作用非常重要，它可以减少网络资源重复请求的次数，不仅减轻了本地的网络负载，而且还减少了爬虫对网站服务器的请求次数，这也是解决反爬虫机制的一个重要手段。

安装 Requests-Cache 可以通过 pip 指令完成，在 CMD 窗口下输入 pip install requests-cache 指令并按回车键，等待安装完成即可。安装成功后进入 Python 交互模式，进一步验证 Requests-Cache 是否安装成功，具体的操作如下：

```
C:\Users\000>python
>>> import requests_cache
>>> requests_cache.__version__
'0.4.13'
```

7.2 在 Requests 中使用缓存

Requests-Cache 遵循 Requests 的使用规则：功能强大并使用简单，整个缓存机制由 install_cache()方法实现。install_cache()方法定义如下所示：

```
install_cache(cache_name='cache',backend=None,expire_after=None,
            allowable_codes=(200,),allowable_methods=('GET',),
            session_factory=CachedSession,**backend_options)
```

install_cache()定义了多个函数参数，每个参数的说明如下：

- cache_name：默认值为 cache，这是对缓存的存储文件进行命名。
- backend：设置缓存的存储机制，默认值为 None，即默认 sqlite 数据库存储。

- expire_after：设置缓存的有效时间，默认值 None，即为永久有效。
- allowable_codes：设置 HTTP 的状态码，默认值为 200。
- allowable_methods：设置请求方式，默认值是只允许 GET 请求才能生成缓存。
- session_factory：设置缓存的执行对象，由 CachedSession 类实现，该类是由 Requests-Cache 定义。
- **backend_options：设置存储配置，若缓存的存储选择 sqlite、redis 或 mongoDB 数据库，则该参数是设置数据库的连接方式。

在实际应用中，install_cache()可以直接使用，无需设置任何参数，因为 Requests-Cache 已对相关的参数设置了默认值，这些默认值基本能满足日常的开发需求。

使用 Requests-Cache 之前，首先创建一个简单的网站系统，这是由 Flask 框架开发的 Web 系统，主要是方便验证 Requests-Cache 的缓存功能。我们需要安装 Flask 框架模块，在 CMD 窗口下输入 pip 指令（pip install flask）并等待安装完成。然后创建 MyFlask.py 文件，并在文件里面编写以下代码：

```
from flask import Flask
# 创建一个 Flask 实例
app = Flask(  name  )

# 设置路由，即 url
@app.route('/')
# url 对应的函数
def hello world():
    # 返回的页面
    return 'Hello World!'

# 程序运行
if   name   == '  main  ':
    app.run()
```

上述代码创建了一个简单的网站首页，网页内容是“Hello World!”。在 PyCharm 或 CMD 窗口下运行 MyFlask.py 文件即可运行一个简单的 Web 系统，网站的后台信息如图 7-1 所示。

```
* Serving Flask app "MyFlask" (lazy loading)
* Environment: production
  WARNING: Do not use the development server in a production environment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

图 7-1　网站的后台信息

使用浏览器访问图上的地址链接即可看到网站的首页，浏览器每次成功访问网站，都会在网站后台出现相关的请求信息。根据这个规则，使用 Requests+Requests-Cache 对网站进行两次访问，查看网站后台请求信息的出现次数。如果请求信息只出现一次，说明爬虫缓存正常使用，反之则说明 Requests-Cache 无法生成缓存。Requests-Cache 的使用方法如下所示：

```
import requests
```

```
import requests_cache
# 使用 install_cache()方法
requests_cache.install_cache()
# 清除已有的缓存
requests_cache.clear()
# 访问自定义的 Web 系统
url = 'http://127.0.0.1:5000/'
# 创建 Session 会话
session = requests.session()
# 执行两次访问
for t in range(2):
    r = session.get(url)
    # from_cache 是 requests_cache 的函数
    # 若输出 True，说明生成缓存。
    print(r.from_cache)
```

运行上述代码，程序会依次输出 False 和 True，False 代表第一次访问还没有生成相关的缓存；True 代表第二次访问就已有相关的缓存数据。同时代码所在的文件路径中会生成 cache.sqlite 文件，这是 sqlite 数据库文件，用于存储缓存信息。此外，网站后台仅有一条请求信息，如图 7-2 所示。

```
 * Serving Flask app "MyFlask" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production environment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [13/Oct/2018 11:55:41] "GET / HTTP/1.1" 200 -
```

图 7-2 请求信息

如果短时间内多次访问网站服务器，很容易遭到服务器的拦截，从而认定这些请求是通过爬虫程序执行，而非人为操作，这是反爬虫常见的机制之一。为了降低访问频率，可以在每个请求之间设置一个 time.sleep()函数，虽然能降低访问频率，但这样处理就显得不太友好。因为两次请求之间，第一次才是真正访问网站后台，而第二次是直接从数据库读取缓存数据，所以这两次请求之间无需设置延时。

那么如何判断这次请求是否已有缓存，每个请求之间应如何合理地设置延时等待？为此，Requests-Cache 可以自定义钩子函数，通过函数去合理判断是否设置延时，函数的定义与使用方法如下：

```
import time
import requests_cache

# 定义钩子函数
def make_throttle_hook(delay=1.0):
    def hook(response, *args, **kwargs):
        # 如果没有缓存，则添加延时
        if not getattr(response, 'from_cache', False):
            print('delayTime')
            time.sleep(delay)
        return response
```

```
    return hook

if __name__ == '__main__':
    requests_cache.install_cache()
    requests_cache.clear()
    # 钩子函数的使用
    s = requests_cache.CachedSession()
    s.hooks = {'response': make_throttle_hook(2)}
    s.get('http://127.0.0.1:5000/')
    s.get('http://127.0.0.1:5000/')
```

从函数 make_throttle_hook 的结构可以看出，这种函数结构其实是一个装饰器的定义过程。也就是说，通过定义装饰器来判断每次请求是否已有缓存数据，从而决定是否设置延时等待。

7.3　缓存的存储机制

Requests-Cache 支持 sqlite、redis 和 mongoDB 数据库存储缓存信息，此外，还可以将缓存存储在计算机的内存中。也就是说 Requests-Cache 支持 4 种不同的存储机制：memory、sqlite、redis 和 mongoDB，4 种存储机制说明如下：

- memory：每次程序运行都会将缓存以字典的形式保存在内存中，程序运行完毕，缓存也随之销毁。
- sqlite：将缓存存储在 sqlite 数据库，这是 Requests-Cache 默认的存储机制。
- redis：将缓存存储在 redis 数据库，通过 redis 模块实现数据库的读写。
- mongoDB：将缓存存储在 mongoDB 数据库，通过 pymongo 模块实现数据库的读写。

在 Requests-Cache 设置不同的存储机制只需对 install_cache()方法的参数 backend 进行设置即可，具体设置如下：

```
import requests_cache
# 设置 memory 存储
requests_cache.install_cache(backend='memory')
# 设置 sqlite 存储
requests_cache.install_cache(backend='sqlite')
# 设置 redis 存储
requests_cache.install_cache(backend='redis')
# 设置 mongo 存储
requests_cache.install_cache(backend='mongo')
```

如果选择 redis 或 mongoDB 作为存储介质，还需要分别安装 redis 模块或 pymongo 模块，这两个模块均可通过 pip 指令安装，同时也要保证本地计算机已安装 redis 或 mongoDB 数据库。

除此之外，Requests-Cache 还提供了其他功能函数，读者可以在 Requests-Cache 的源码文件（Lib\site-packages\requests_cache\core.py）找到相关函数以及说明。

7.4 本章小结

Requests-Cache 是 Requests 模块的一个扩展功能，它是根据 Requests 的发送请求来生成相应的缓存数据，其作用非常重要，可以减少网络资源重复请求的次数，不仅减轻了本地的网络负载，而且还可以减少爬虫对网站服务器的请求次数，这也是解决反爬虫机制的一个重要手段。

整个缓存机制由 install_cache()方法实现，该方法的参数说明如下。

- cache_name：默认值为 cache，这是对缓存的存储文件进行命名。
- backend：设置缓存的存储机制，默认值为 None，即默认 sqlite 数据库存储。
- expire_after：设置缓存的有效时间，默认值 None，即为永久有效。
- allowable_codes：设置 HTTP 的状态码，默认值为 200。
- allowable_methods：设置请求方式，默认值是只允许 GET 请求才能生成缓存。
- session_factory：设置缓存的执行对象，由 CachedSession 类实现，该类是由 Requests-Cache 定义。
- **backend_options：设置存储配置，若缓存的存储选择 sqlite、redis 或 mongoDB 数据库，则该参数是设置数据库的连接方式。

Requests-Cache 支持 4 种不同的存储机制：memory、sqlite、redis 和 mongoDB，4 种存储机制说明如下。

- memory：每次程序运行都会将缓存以字典的形式保存在内存中，程序运行完毕，缓存也随之销毁。
- sqlite：将缓存存储在 sqlite 数据库，这是 Requests-Cache 默认的存储机制。
- redis：将缓存存储在 redis 数据库，通过 redis 模块实现数据库的读写。
- mongoDB：将缓存存储在 mongoDB 数据库，通过 pymongo 模块实现数据库的读写。

第 8 章

爬虫库 Requests-HTML

8.1 简介及安装

Requests-HTML 是在 Requests 的基础上进一步封装，两者都是由同一个开发者开发。Requests-HTML 除了包含 Requests 的所有功能之外，还新增了数据清洗和 Ajax 数据动态渲染。

数据清洗是由 lxml 和 PyQuery 模块实现，这两个模块分别支持 XPath Selectors 和 CSS Selectors 定位，通过 XPath 或 CSS 定位，可以精准地提取网页里的数据。

Ajax 数据动态渲染是将网页的动态数据加载到网页上再抓取。网页数据可以使用 Ajax 向服务器发送 HTTP 请求，再由 JavaScript 完成数据渲染，如果直接向网页的 URL 地址发送 HTTP 请求，并且网页的部分数据是来自 Ajax，那么，得到的网页信息就会有所缺失。而 Requests-HTML 可以将 Ajax 动态数据加载到网页信息，无需爬虫开发者分析 Ajax 的请求信息。

Requests-HTML 的安装可使用 pip 指令完成，但 Requests-HTML 只支持 Python 3.6 以上的版本。本书以 Python 3.7 为例，在 CMD 窗口输入安装指令 pip install requests-html，等待安装完成即可。

在 CMD 窗口进入 Python 交互模式，通过导入 requests-html 模块并输出模块里的属性 DEFAULT_URL 的属性值，从而验证 requests-html 模块是否安装成功，如下所示：

```
C:\Users\000>python
>>> import requests html
>>> requests html.DEFAULT URL
'https://example.org/'
```

8.2 请 求 方 式

Requests-HTML 向网站发送请求的方法是来自 Requests 模块，但是 Requests-HTML 只能使用 Requests 的 Session 模式，该模式是将请求会话实现持久化，使这个请求保持连接状态。Session 模式好比我们在打电话的时候，只要双方没有挂断电话，就会一直保持一种会话（连接）状态。Session 模式对 HTTP 的 GET 和 POST 请求也是由 get()和 post()方法实现，具体的使用方法如下：

```
from requests_html import HTMLSession
# 定义会话 Session
session = HTMLSession()
url = 'https://movie.douban.com/'
# 发送 GET 请求
r = session.get(url)
# 发送 POST 请求
r = session.post(url, data={})
# 输出网页的 URL 地址
print(r.html)
```

上述代码分别对同一个 URL 使用 get()和 post()方法，由于 get()和 post()方法都来自 Requests 模块，因此还可以对这两个方法设置相关的参数，如请求参数、请求头、Cookies、代理 IP 以及证书验证等。

Requests-HTML 在请求过程中还做了优化处理，如果没有设置请求头，Requests-HTML 就会默认使用源码里所定义的请求头以及编码格式。在 Python 的安装目录下打开 Requests-HTML 的源码文件（\Lib\site-packages\requests_html.py），定义了属性 DEFAULT_ENCODING 和 DEFAULT_USER_AGENT，分别对应编码格式和 HTTP 的请求头，如图 8-1 所示。

```
DEFAULT_ENCODING = 'utf-8'
DEFAULT_URL = 'https://example.org/'
DEFAULT_USER_AGENT = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_6) AppleWebKit/603.3.8 (KHTML, like Gecko) Version/10.1.2 Safari/603.3.8'
DEFAULT_NEXT_SYMBOL = ['next', 'more', 'older']
```

图 8-1 默认属性

8.3 数 据 清 洗

Requests-HTML 不仅优化了请求过程，还提供了数据清洗的功能，而 Requests 模块只提供请求方法，并不提供数据清洗，这也体现了 Requests-HTML 的一大优点。使用 Requests 开发的爬虫，数据清洗需要调用其他模块实现，而 Requests-HTML 则将两者结合在一起。

Requests-HTML 提供了各种各样的数据清洗方法，比如网页里的 URL 地址、HTML 源码内容、文本信息等，使用方法如下所示：

```
from requests_html import HTMLSession
# 定义会话 Session
session = HTMLSession()
```

```
url = 'https://movie.douban.com/'
# 发送 GET 请求
r = session.get(url)
# 输出网页的 URL 地址
print(r.html)
# 输出网页里全部 URL 地址
print(r.html.links)
# 输出网页里精准的 URL 地址
print(r.html.absolute links)
# 输出网页的 HTML 信息
print(r.text)
# 输出网页的全部文本信息，即去除 HTML 代码
print(r.html.text)
```

上述代码只是提取了网站的基本信息，如果想要精准地提取某个数据，可以使用 find()、xpath()、search()和 search_all()方法实现。首先了解这 4 种方法的定义及相关的参数说明：

```
#定义
find(selector, containing, clean, first,  encoding)
#参数说明
selector：使用 CSS Selector 定位网页元素。
containing：字符串类型，默认值为 None，通过特定文本查找网页元素。
clean：是否清除 HTML 的<script>和<style>标签，默认值为 False。
first：是否只查找第一个网页元素，默认值为 False 即查找全部元素。
_encoding：设置编码格式，默认值为 None。

#定义
xpath(selector, clean, first,  encoding)
#参数说明
selector：使用 XPath selector 定位网页元素。
clean：是否清除 HTML 的<script>和<style>标签，默认值为 False。
first：是否只查找第一个网页元素，默认值为 False 即查找全部元素。
 encoding：设置编码格式，默认值为 None。

#定义
search(template)
#参数说明
template：通过元素内容查找第一个元素

#定义
search_all(template)
#参数说明
template：通过元素内容查找全部元素
```

以豆瓣电影的网页为例，在浏览器中打开豆瓣电影的网页并使用开发者工具分析网页信息，分别提取电影名与评分，网页元素信息如图 8-2 所示。



图 8-2　数据精准提取

从图 8-2 中发现，电影名在标签<li class="title">里，评分在标签<li class="rating">里，因此上述 4 种定位方法的使用如下所示：

```
from requests_html import HTMLSession
# 定义会话 Session
session = HTMLSession()
url = 'https://movie.douban.com/'
# 发送 GET 请求
r = session.get(url)

# 通过 CSS Selector 定位 li 标签，".title"代表 class 属性
# first=True 代表获取第一个元素
print(r.html.find('li.title', first=True).text)
# 输出当前标签的属性值
print(r.html.find('li.title', first=True).attrs)
print('______分割线______')

# 查找特定文本的元素
# 如果元素所在的 HTML 里含有 containing 的属性值即可提取
for name in r.html.find('li', containing='超能'):
    # 输出电影名
    print(name.text)
print('______分割线______')

# 查找全部电影名
for name in r.html.find('li.title'):
    # 输出电影名
    print(name.text)
    # 输出电影名所在标签的属性值
    print(name.attrs)
print('______分割线______')

# 通过 XPath Selector 定位 ul 标签
x = r.html.xpath('//*[@id="screening"]/div[2]/ul')
for name in x:
    print(name.text)
print('______分割线______')
```

```
# search()通过关键字查找内容
# 一个{}代表一个内容，内容可为中文或英文等
print(r.html.search('古剑奇谭{}{}'))
print('______分割线______')

# search_all()通过关键字查找整个网页符合的内容
# 一个{}代表一个内容，内容可为中文或英文等
print(r.html.search_all('古剑奇谭{}{}'))
```

如果使用 XPath Selector 或 CSS Selector 实现元素定位，需要掌握 XPath 或 CssSelector 语法。这两者的语法本书不做详细介绍，有兴趣的读者可自行查阅相关资料。

8.4　Ajax 动态数据抓取

如果使用 Requests-HTML 请求网页地址，相应的响应内容与开发者工具的 Doc 选项卡的响应内容是一致的。如果网页数据是通过 Ajax 请求并由 JavaScript 渲染到网页上，还需要使用 Requests-HTML 模拟 Ajax 请求来获取网页数据。

对于爬虫开发者来说，模拟 Ajax 请求是一件相当痛苦的事情，比如构建请求参数，请求参数的构建方式繁多而复杂，这非常考验开发者对网站的分析能力。以 QQ 音乐的歌手列表页为例，每位歌手的名字都是由 Ajax 加载到网页上，如图 8-3 所示。

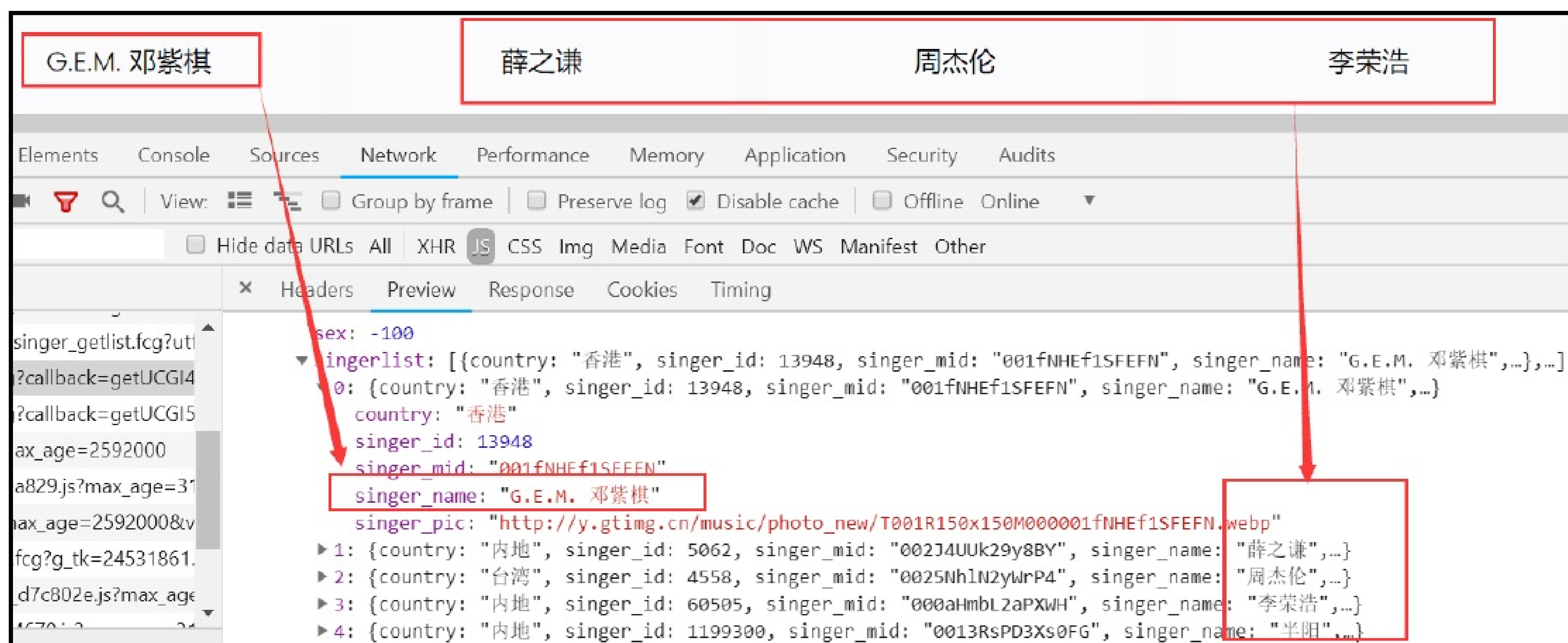


图 8-3　歌手信息分析

为了降低开发难度，Requests-HTML 提供了 Ajax 加载功能，加载后的网页信息与开发者工具的 Elements 选项卡的网页信息是一致的。这个加载功能是通过调用谷歌的 Chromium 浏览器实现的，Chromium 是谷歌为发展 Chrome 而开启的计划，它可以理解为 Chrome 的工程版或实验版，新功能都会率先在 Chromium 上实现，待验证后才会应用在 Chrome 上。

Ajax 加载功能由 render()方法实现，初次使用 render()方法会自动下载 Chromium 浏览器，下载 Chromium 浏览器必须保证当前网络能正常访问谷歌首页，否则无法下载。此外，还可以直接下载 Chromium 浏览器，并将浏览器放置在 C 盘的用户文件夹，如图 8-4 所示。

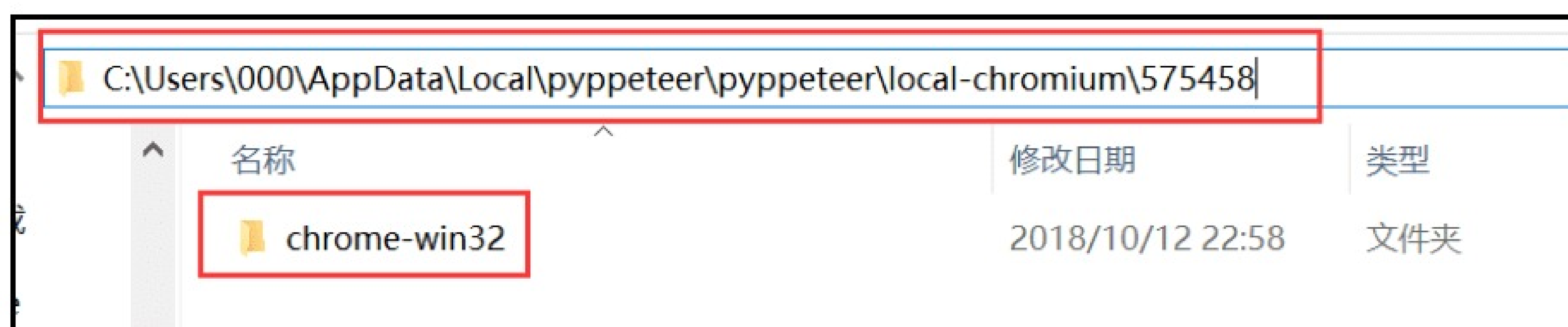


图 8-4 配置 Chromium 浏览器

在图 8-4 上的文件路径中，只有“000”是变化的，不同的电脑有不同的命名；而 chrome-win32 文件夹的命名也是固定的，该文件夹里存放了 Chromium 浏览器的相关文件和应用程序。如果是通过下载方式就无需手动配置文件路径，Requests-HTML 会将下载后的 Chromium 浏览器自动配置到相应的文件路径，如图 8-5 所示。

```
[W:pyppeteer.chromium_downloader] start chromium download.
Download may take a few minutes.
100%|███████████| 133194757/133194757 [04:41<00:00, 472789.53it/s]
[W:pyppeteer.chromium_downloader]
chromium download done.
[W:pyppeteer.chromium_downloader] chromium extracted to: C:\Users\000\AppData\Local\pyppeteer\pyppeteer\local-chromium\575458
```

图 8-5 Chromium 浏览器自动配置

完成了 Chromium 浏览器配置，可以编写以下代码来实现 Requests-HTML 的 Ajax 加载功能：

```
from requests html import HTMLSession
url = 'https://y.qq.com/portal/singer list.html'
session = HTMLSession()
r = session.get(url)
#使用 Chromium 浏览器加载网页
r.html.render()
#定位歌手姓名
singer = r.html.find('h3.singer list  title')
#输出歌手姓名
for i in singer:
    print(i.text)
```

在 PyCharm 里运行上述代码，可以看到程序将歌手姓名逐一输出。虽然运行速度比模拟 Ajax 请求的速度较慢，但可以大大降低开发难度，运行结果如图 8-6 所示。

```
G.E.M. 邓紫棋
薛之谦
周杰伦
李荣浩
半阳
林俊杰
白小白
毛不易
庄心妍
黑龙
```

图 8-6 运行结果

8.5　本 章 小 结

Requests-HTML 是在 Requests 的基础上进一步封装，这两个爬虫库都是由同一个开发者开发的。Requests-HTML 除了包含 Requests 的所有功能之外，还新增了数据提取和 Ajax 数据动态渲染。

Requests-HTML 只能使用 Requests 的 Session 模式，该模式是将请求会话实现持久化，可使这个请求保持连接状态。Session 模式好比我们打电话，只要双方没有挂断电话，就会一直保持一种会话（连接）状态。

数据提取是由 lxml 和 PyQuery 模块实现，这两个模块分别支持 XPath Selectors 和 CSS Selectors 定位，通过 XPath 或 CSS 定位，可以精准地提取网页里的数据。

Ajax 数据动态渲染是将网页的动态数据加载到网页上再抓取，它是由 Requests-HTML 的 render()方法实现，通过调用 Chromium 浏览器来加载 Ajax 功能，从而实现网页信息加载。

第 9 章

网页操控与数据爬取

9.1 了解 Selenium

Selenium 是一个用于网站应用程序自动化的工具。它可以直接运行在浏览器中，就像真正的用户在操作一样。它支持的浏览器包括 IE、Mozilla Firefox、Safari、Google Chrome 和 Opera 等，同时支持多种编程语言，如.Net、Java、Python 和 Ruby 等。

Jason Huggins 在 2004 年发起了 Selenium 项目，这个项目主要是为了不想让自己的时间浪费在无聊的重复性工作中。当时测试的浏览器都支持 JavaScript，因此 Jason 和他所在的团队采用 JavaScript 编写一种测试工具来验证浏览器页面的行为。这个 JavaScript 类库就是 Selenium core，同时也是 selenium RC、Selenium IDE 的核心组件，Selenium 由此诞生。关于 Selenium 的命名比较有意思，当时 QTP mercury 是主流的商业自动化工具，这是化学元素汞（俗称水银），而 Selenium 是开源自动化工具，是化学元素硒，硒可以对抗汞。

从 Selenium 诞生至今一共发展了 3 个版本：Selenium 1.0、Selenium 2.0 和 Selenium 3.0。每个版本的更新都有一些变化，下面大概了解一下各个版本的信息。

Selenium 1.0：主要由 Selenium IDE、Selenium Grid 和 Selenium RC 组成。Selenium IDE 是嵌入到浏览器的一个插件，用于实现简单的浏览器操作的录制与回放功能；Selenium Grid 是一种自动化的辅助工具，通过利用现有的计算机基础设施，能加快网站自动化操作；Selenium RC 是 Selenium 家族的核心部分，支持多种不同开发语言编写的自动化脚本，可通过 Selenium RC 的服务器作为代理服务器去访问网站应用，从而达到自动化目的。

Selenium 2.0：在 1.0 版本的基础上结合了 Webdriver。Selenium 通过 Webdriver 直接操控网站应用，解决了 Selenium 1.0 存在的缺点。WebDriver 针对各个浏览器而开发，取代了网站应用的 JavaScript。目前大部分自动化技术都是以 Selenium 2.0 为主，这也是本章主要讲述的内容。

Selenium 3.0：这个版本做了不大不小的更新。如果是使用 Java 开发只能在 Java 8 以上的开发环境，如果以 IE 浏览器作为自动化浏览器，浏览器必须在 IE 9 版本或以上。

从 Selenium 的各个版本信息可以了解到，它必须在浏览器的基础上才能实现自动化。目前浏

览器的种类繁多，比如搜狗浏览器、QQ 浏览器和百度浏览器等，这些浏览器大多数是在 IE 内核、Webkit 内核或 Gecko 内核的基础上开发而成的。为了统一浏览器的使用，Selenium 主要支持 IE、Mozilla Firefox、Safari、Google Chrome 和 Opera 等国际性主流浏览器。

Selenium 发展至今，不仅在自动化测试和自动化流程开发的领域上占据着重要的位置，而且在网络爬虫上也广泛被使用。

9.2　安装 Selenium

由于 Selenium 支持多种浏览器，本书以 Google Chrome 作为讲述对象。搭建 Selenium 开发环境需要安装 Selenium 库并且配置 Google Chrome 的 WebDriver。安装 Selenium 库可以使用 pip 指令完成，具体的安装指令如下所示：

```
pip install selenium
```

Selenium 安装完成后，我们在 CMD 环境下验证 Selenium 是否安装成功。在 CMD 里输入“python”并按回车，就会进入 Python 的交互模式。在交互模式下依次输入以下代码：

```
>>> import selenium
>>> selenium.__version__
'3.14.0'
```

从上述的代码的可以，在 Python 的交互模式下成功导入 Selenium 库，并且当前 Selenium 库的版本信息为 3.14.0。Selenium 的安装相对较为简单，接下来是安装 Google Chrome 的 WebDriver。首先打开 Google Chrome 并查看当前的版本信息。在浏览器找到“自定义及控制 Google Chrome”→“帮助(E)”→“关于 Google Chrome(G)”按钮，如图 9-1 所示。



图 9-1　浏览器版本查看方法

除了上述方法之外，还可以在浏览器的地址上直接输入 chrome://settings/help 并按回车即可查看浏览器的版本信息。版本信息如图 9-2 所示。



图 9-2　浏览器版本信息

从图 9-2 中可以得知，当前 Google Chrome 的版本为 68，根据版本信息找到与之对应的 WebDriver 版本，Google Chrome 与 WebDriver 版本对照表如表 9-1 所示。

表 9-1

chromedriver 版本（WebDriver）	Google Chrome 版本
v2.40	v66-68
v2.39	v66-68
v2.38	v65-67
v2.37	v64-66
v2.36	v63-65
v2.35	v62-64
v2.34	v61-63
v2.33	v60-62
v2.32	v59-61
v2.31	v58-60

根据浏览器的版本号与对照表可以知道，chromedriver（WebDriver）版本号应为 v2.40 或 v2.39。在浏览器上访问 http://npm.taobao.org/mirrors/chromedriver/并找到 v2.40 所在的位置，进入 v2.40 并单击 chromedriver_win32.zip 的下载链接。把下载的 chromedriver_win32.zip 进行解压，然后双击运行 chromedriver.exe，查看 chromedriver 的版本信息，如图 9-3 所示。

```
Starting ChromeDriver 2.40.565498 (ea082db3280dd6843ebfb08a625e3eb905c4f5ab) on port 9515
Only local connections are allowed.
```

图 9-3　chromedriver 的版本信息

确认 chromedriver 的版本信息无误之后，我们将 chromedriver.exe 直接放置在 Python 的安装目录，比如本书的 Python 安装目录在 E:\Python，如图 9-4 所示。



图 9-4　chromedriver.exe 存放位置

完成 Selenium 库的安装以及 chromedriver 的配置后，在 PyCharm 里创建一个 test.py 文件，编写以下代码来验证 Selenium 是否能自动启动并控制 Google Chrome。代码如下：

```
# 导入 Selenium 的 webdriver 类
from selenium import webdriver
# 设置变量 url，用于浏览器访问
url = 'https://www.baidu.com/'
# 将 webdriver 类实例化，将浏览器设定为 Google Chrome
# 参数 executable_path 是设置 chromedriver 的路径
path='E:\\Python\\chromedriver.exe'
browser = webdriver.Chrome(executable_path=path)
# 打开浏览器并访问百度网址
browser.get(url)
```

上述代码分为三个步骤：导入 Selenium 库的 webdriver 类、webdriver 类实例化并指定浏览器、打开浏览器访问网址。如果 chromedriver.exe 是存放在 Python 的安装目录中，在 webdriver 类实例化的时候，可以无需设置参数 executable_path；但如果 chromedriver.exe 是存放在其他目录，在实例化的时候要设置参数 executable_path 来指向 chromedriver.exe 所在的位置。上述代码运行后，程序会自动打开一个新的 Google Chrome，如图 9-5 所示。



图 9-5　Selenium 控制 Google Chrome

此外，Selenium 还可以控制其他浏览器，在执行程序之前，记得配置浏览器的 WebDriver，配置方法与配置 Google Chrome 的大同小异。首先通过浏览器版本确认 WebDriver 的版本，然后下载

WebDriver 并存放在 Python 的安装目录。以 IE 和 Mozilla Firefox（火狐）为例，两者的 WebDriver 配置过程就不作详细讲述，此处只列出 Selenium 的具体代码，如下所示：

```
# 启动火狐浏览器
from selenium import webdriver
browser = webdriver.Firefox()
browser.get('http://www.baidu.com/')

# 启动 IE 浏览器
from selenium import webdriver
browser = webdriver.Ie()
browser.get('http://www.baidu.com/')
```

9.3 网页元素定位

Selenium 抓取网页信息是在谷歌开发者工具的 Elements 选项卡里，本节主要讲述如何将网页元素告知 Selenium，并让它自动操控网页及读取数据。Selenium 定位网页元素主要通过元素的属性值或者元素在 HTML 里的路径位置，定位方式一共有 8 种，如下所示：

```
# 通过属性 id 和 name 来实现定位
find element by id()
find element by name()

# 通过 HTML 标签类型和属性 class 实现定位
find element by class name()
find element by tag name()

# 通过标签值实现定位，partial link 用于模糊匹配。
find element by link text()
find element by partial link text()

# 元素的路径定位选择器
find element by xpath()
find_element_by_css_selector()
```

我们将 8 种定位方式分为 4 组，分组标准以每种定位方式的优缺点来进行划分。具体的说明如下：

（1）find_element_by_id 和 find_element_by_name 分别通过元素属性 id 和 name 的属性值来定位。如果被定位的元素不存在属性 id 或 name，则无法使用这种定位方式。通常情况下，一个网页中，元素的 id 或 name 的属性值是唯一的，如果多个元素的 id 或 name 相同，这种定位方式只能定位第一个元素。

（2）find_element_by_class_name 和 find_element_by_tag_name 分别通过元素属性 class 和元素标签类型进行定位。在一个网页里，属性 class 的属性值可以被多个元素使用，同一个元素标签也可以多次使用，正因如此，这两种定位方式只能定位符合条件的第一个元素。

（3）find_element_by_link_text 和 find_element_by_partial_link_text 是根据标签值进行定位。比如单击豆瓣电影网的排行榜，通过网页的文字来对元素进行定位。若网页中的文字并不是唯一的，

那么 Selenium 也是默认定位第一个符合条件的元素。

（4）find_element_by_xpath 和 find_element_by_css_selector 是由 xpath 和 css_selector 实现定位，两者是一个定位选择器，通过标签的路径来实现定位。标签的路径是指当前标签在整个 HTML 代码里的代码位置，比如<body>里的第二个<div>标签，<div>又嵌套<p>标签，那么<p>的路径为 body → div[1] → p。这种定位方式相对前面的定位较为精准，因为每个标签的路径都是唯一的。

我们以豆瓣电影网为例，具体讲述 8 种定位方式的使用，代码如下：

```
from selenium import webdriver
url = 'https://movie.douban.com/'
driver = webdriver.Chrome()
driver.get(url)
# 定位
driver.find_element_by_id('inp-query').send_keys('红海行动')
driver.find_element_by_name('search_text').send_keys('我不是药神')
```

find_element_by_id 和 find_element_by_name 都是定位网页的搜索框，并在搜索框里输入文本信息。文本框的元素信息如图 9-6 所示。



图 9-6　搜索框元素信息

```
class_name = driver.find_element_by_class_name('nav-items').text
tag_name = driver.find_element_by_tag_name('div').text
print('由class_name定位：', class_name)
print('由tag_name定位：', tag_name)
```

上述两种方式分别定位不同的网页元素。class_name 是定位 class 属性值为 nav-items 的标签，tag_name 是定位 HTML 里面第一个<div>标签，两者定位元素后，再使用 text 方法来获取元素值并输出。元素信息如图 9-7 所示。

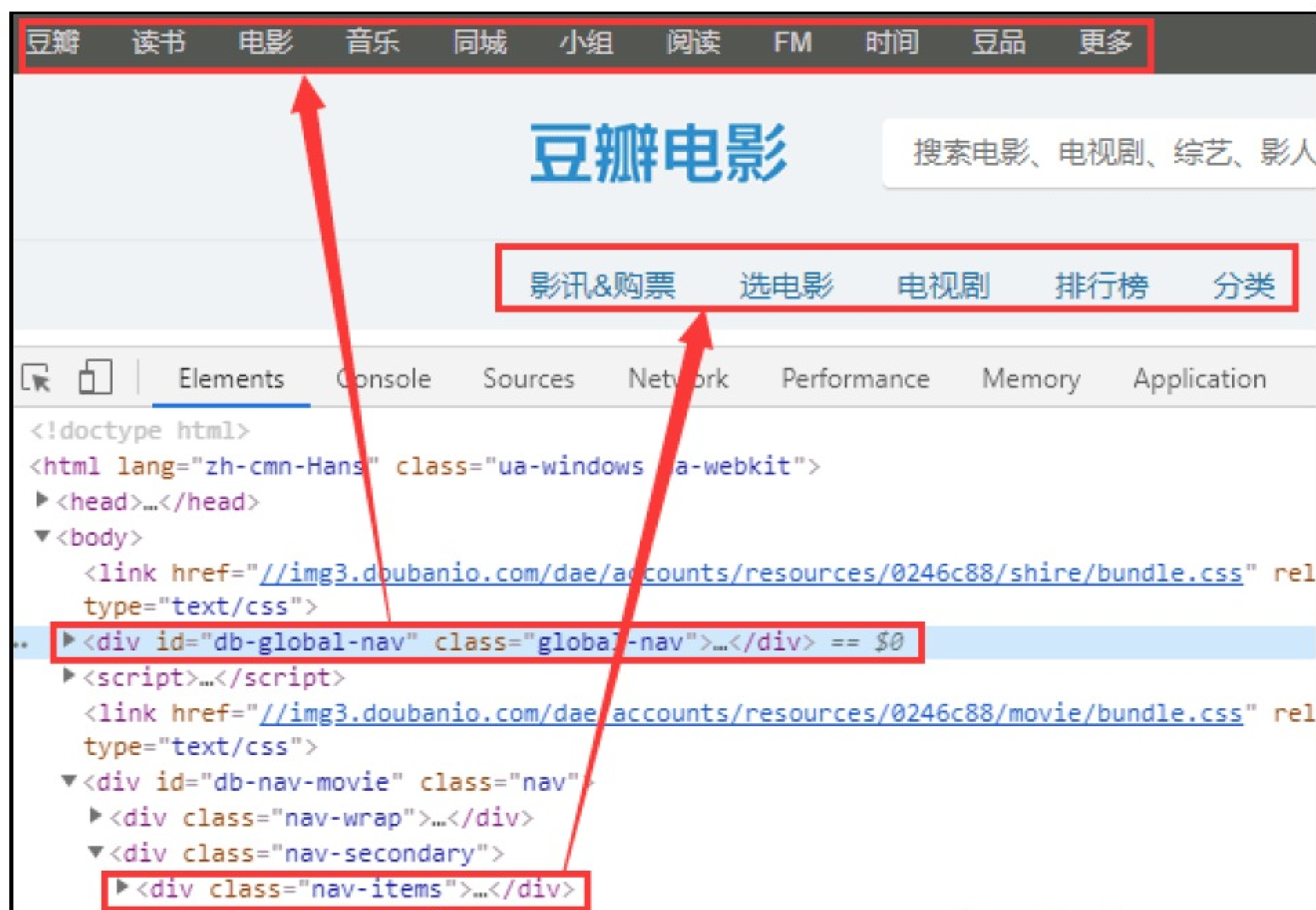


图 9-7 class_name 和 tag_name 定位元素

```
    link_text = driver.find_element_by_link_text('排行榜').text
    partial_text = driver.find_element_by_partial_link_text('部正在热映').text
    print('由 link_text 定位：', link_text)
    print('由 partial_link_text 定位：', partial_text)
```

上述代码是将网页中含有“排行榜”和“部正在热映”的内容进行定位，“排行榜”在网页中只出现一次，link_text 是对内容进行精准定位，比如网页中出现“排行榜”和“国语排行榜”，link_text 只能定位到“排行榜”。而“部正在热映”是网页内容“全部正在热映»”的部分内容，partial_link_text 表示可以进行模糊匹配，所以 Selenium 会自动定位“全部正在热映»”这个元素。如图 9-8 所示。



图 9-8 link_text 和 partial_link_text 定位元素

```
    xpath = driver.find_element_by_xpath('//*
    [@id="db-nav-movie"]/div[1]/div/div[1]/a').text
```

```
    selector = driver.find_element_by_css_selector('#db-nav-movie
    > div.nav-wrap > div > div.nav-logo > a').text
    print('由 xpath 定位：', xpath)
    print('由 css_selector 定位：', selector)
```

例子中的定位选择器 xpath 和 css_selector 都是定位 class 属性值为 nav-logo 的<div>标签里的<a>标签，然后再获取该标签的值并输出。xpath 和 css_selector 的语法编写规则各不相同，一般情况下，在 Google Chrome 里可以快速获取两者的语法。首先在 Google Chrome 的 Elements 标签页里，找到某个元素的位置，然后右击选择“Copy”，最后选择“Copy Xpath”或“Copy selector”即可获取相应的语法。如图 9-9 所示。

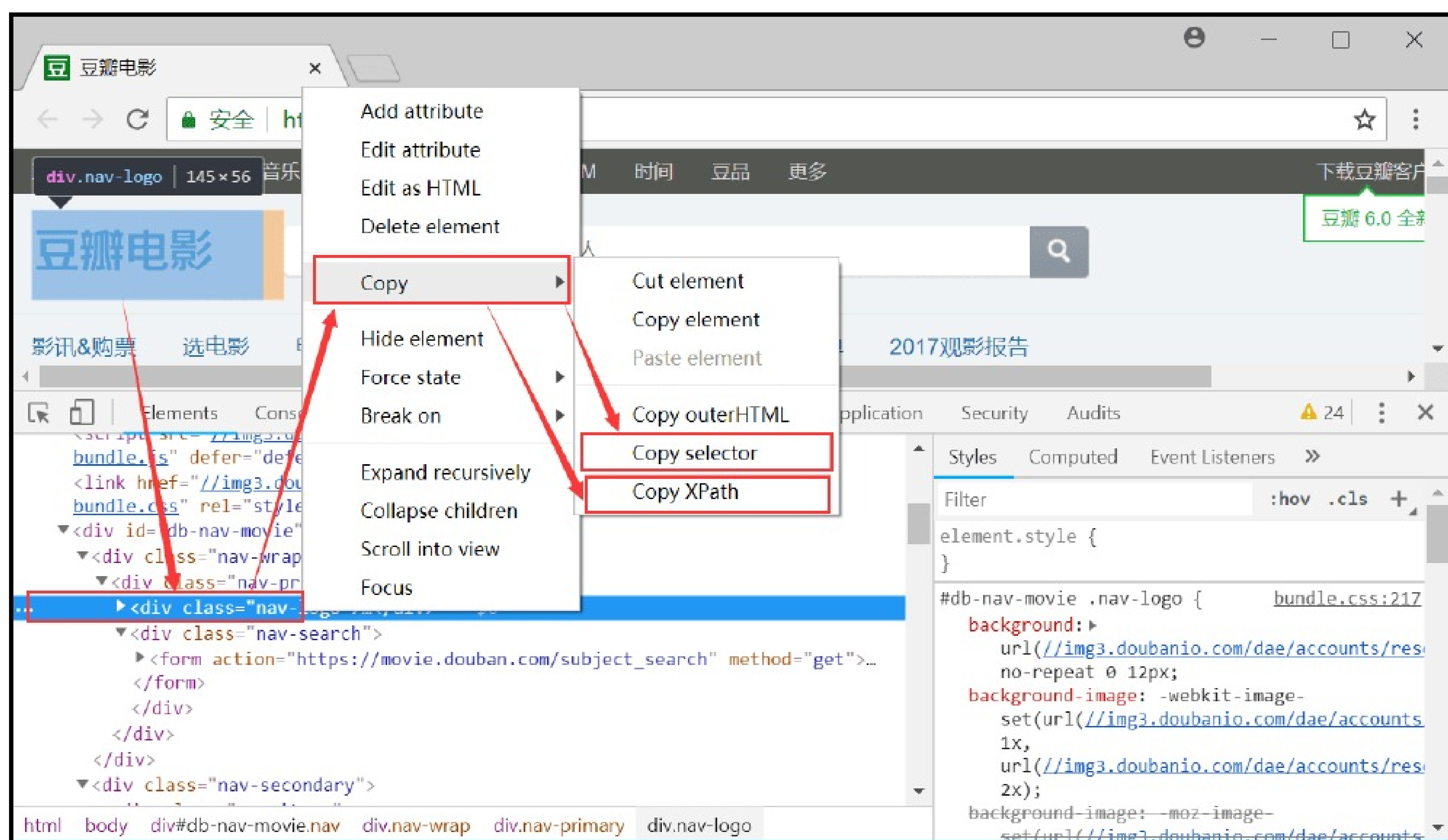


图 9-9　xpath 和 css_selector 语法获取

上述 8 种定位方式只能定位的第一个元素，如果有多个相同的元素，并且想全部获取，可以使用以下定位方式：

```
    find_elements_by_id()
    find_elements_by_name()
    find_elements_by_class_name()
    find_elements_by_tag_name()
    find_elements_by_link_text()
    find_elements_by_partial_link_text()
    find_elements_by_xpath()
    find_elements_by_css_selector()
```

这 8 种定位方式与上述的定位方式非常相似，两者的唯一不同就是 elements 和 element。前者是定位全部符合条件的元素，后者只是获取第一个符合条件的元素。

在上述中所提及到的 xpath 和 css_selector 的语法编写，有兴趣的读者可以自行查阅相关的资料，进一步了解两者的语法编写规则。

9.4 网页元素操控

操控网页元素在网页元素定位后才能执行，Selenium 可以模拟任何操作，比如单击、右击、拖拉、滚动、复制粘贴或者文本输入等等，操作方式可分为三大类：常规操作、鼠标事件操作和键盘事件操作。

常规操作包含文本清除、文本输入、单击元素、提交表单、获取元素值等。以 QQ 音乐注册为例（https://ssl.zc.qq.com/v3/index-chs.html?from=pt），具体的使用方式如下：

```
from selenium import webdriver
url = 'https://ssl.zc.qq.com/v3/index-chs.html?from=pt'
driver = webdriver.Chrome()
driver.get(url)
# 输入名字和密码
driver.find_element_by_id('nickname').send_keys('pythonAuto')
driver.find_element_by_id('password').send_keys('pythonAuto123')
# 获取手机号码下方的 tips 内容
tipsValue = driver.find_element_by_xpath(
'//div[3]/div[2]/div[1]/form/div[7]/div').text
print(tipsValue)
# 勾选同时开通 QQ 空间
driver.find_element_by_class_name('checkbox').click()
# 点击"注册"按钮
driver.find_element_by_id('get_acc').submit()
```

上述例子对网页的昵称和密码的文本框执行文本输入、获取手机号码下方的 tips 内容、勾选“同时开通 QQ 空间”选项和单击“注册”按钮，4 种操作分别由 send_keys、text、click 和 submit 方法实现。其中 click 和 submit 在某些情况下可以相互使用，submit 只用于表单的提交按钮；click 是强调事件的独立性，可用于任何按钮。此外，我们还列出一些实际开发中常见的操作方式：

```
# 清空 X 标签的内容
driver.find_element_by_id('X').clear()
# 获取元素在网页中的坐标位置，坐标格式：{'y': 19, 'x': 498}
location = driver.find_element_by_id('X').location
# 获取元素的某个属性值，如获取 X 标签的 id 属性值
attribute = driver.find_element_by_id('X').get_attribute('id')
# 判断 X 元素在网页上是否可见，返回值为 True 或 False
result = driver.find_element_by_id('X').is_displayed()
# 判断 X 元素是否被选，通常用于 checkbox 和 radio 标签，返回值为 True 或 False
result = driver.find_element_by_id('X').is_selected()
""" select 标签的选值 """
from selenium.webdriver.support.select import Select
# 根据下拉框的索引来选取
Select(driver.find_element_by_id('X')).select_by_index('2')
# 根据下拉框的 value 属性来选取
Select(driver.find_element_by_id('X')).select_by_index('Python')
# 根据下拉框的值来选取
Select(driver.find_element_by_id('X')).select_by_visible_text('Python')
```

上述是元素常规操作方法，接着讲述鼠标事件操作方法，鼠标事件操作由 Selenium 的 ActionChains 类来实现。ActionChains 类定义了多种鼠标操作方法，具体的操作方法说明如表 9-1 所示。

表 9-1　ActionChains 类的鼠标操作方法

操作方法	说明	示例
perform	执行鼠标事件	click(element).perform() click 是鼠标单击事件 perform 是执行这个单击事件
reset_actions	取消鼠标事件	click(element).reset_actions() click 是鼠标单击事件 reset_actions 是取消单击事件
click	鼠标单击	click(element) element 是某个元素对象
click_and_hold	长按鼠标左键	click_and_hold(element) element 是某个元素对象
context_click	长按鼠标右键	context_click(element) element 是某个元素对象
double_click	鼠标双击	double_click(element) element 是某个元素对象
drag_and_drop	对元素长按左键并移动到另一个元素的位置后释放鼠标左键	drag_and_drop(element, element1) element 是某个元素对象 element1 是目标元素对象
drag_and_drop_by_offset	对元素长按左键并移动到指定的坐标位置	drag_and_drop_by_offset(element, x, y) element 是某个元素对象 x 是偏移的 x 坐标 y 是偏移的 y 坐标
key_down	对元素长按键盘中的某个按键	key_down(Keys.CONTROL,element) Keys.CONTROL 是由 Keys 定义的键盘事件 element 是某个元素对象
key_up	对元素释放键盘中的某个按键	key_up(Keys.CONTROL,element) Keys.CONTROL 是由 Keys 定义的键盘事件 element 是某个元素对象
move_by_offset	对当前鼠标所在位置进行偏移	move_by_offset(x, y) x 是偏移的 x 坐标 y 是偏移的 y 坐标
move_to_element	将鼠标移动到某个元素所在的位置	move_to_element(element) element 是某个元素对象

（续表）

操作方法	说明	示例
move_to_element_with_offset	将鼠标移动到某个元素并偏移一定的位置	move_to_element_with_offset(element, x, y) element 是某个元素对象 x 是偏移的 x 坐标 y 是偏移的 y 坐标
pause	设置暂停执行时间	pause(1000)
release	释放鼠标长按操作	release(element) element 是某个元素对象 如果 element 为空，对当前鼠标的位置长按操作进行释放
send_keys	执行文本输入	send_keys(value) value 是输入的内容
send_keys_to_element	对当前元素执行文本输入	send_keys_to_element(element, value) element 是某个元素对象 value 是输入的内容

上表讲述了各种鼠标事件操作，这些方法都是在 ActionChains 类所定义的类方法，若想使用这些操作方法，必须将 ActionChains 类实例化后才能调用。以 B 站的登录页面为例，通过鼠标操作方法去双击网页中的“登录”标题以及拖拉验证滑条。具体代码如下：

```
from selenium import webdriver
from selenium.webdriver.common.action chains import ActionChains
import time
url = 'https://passport.bilibili.com/login'
driver = webdriver.Chrome()
driver.get(url)
# 双击登录
element = driver.find element by class name('tit')
ActionChains(driver).double click(element).perform()
#设置延时，否则会导致操作过快
time.sleep(3)
# 拖拉滑条
element = driver.find element by class name('gt slider knob,gt show')
ActionChains(driver).drag_and_drop_by_offset(element, 100, 0).perform()
```

上述代码中，首先将 ActionChains 实例化，实例化的时候传入 driver 对象。driver 是 chromedriver 打开的浏览器对象，这是告诉 ActionChains 的操作浏览器对象是 driver。实例化之后就可以直接调用鼠标事件操作方法，这些方法需要传入 element 对象，element 是网页中某个标签元素。最后再调用 perform 方法，这是一个执行命令，因为鼠标操作可以拖拉、长按鼠标的左键或右键，这是一个持久性的操作，而调用 perform 方法可以让这个鼠标操作马上执行。

最后讲述键盘事件操作，它是模拟人为按下键盘的某个按键，主要通过 send_keys 方法来实现。在上述例子中，send_keys 用于文本内容的输入，而本例是通过 send_keys 来触发键盘按钮。以百度搜索为例，利用键盘的快捷键实现搜索内容的变换。具体代码如下所示：

```
from selenium import webdriver
```

```
from selenium.webdriver.common.keys import Keys
import time

driver = webdriver.Chrome()
driver.get("http://www.baidu.com")

# 获取输入框标签对象
element = driver.find element by id('kw')
# 输入框输入内容
element.send_keys("Python 你")
time.sleep(2)

# 删除最后的一个文字
element.send_keys(Keys.BACK_SPACE)
time.sleep(2)

# 添加输入空格键+“教程”
element.send keys(Keys.SPACE)
element.send keys("教程")
time.sleep(2)

# ctrl+a 全选输入框内容
element.send keys(Keys.CONTROL, 'a')
time.sleep(2)

# ctrl+x 剪切输入框内容
element.send keys(Keys.CONTROL, 'x')
time.sleep(2)

# ctrl+v 粘贴内容到输入框
element.send keys(Keys.CONTROL, 'v')
time.sleep(2)

# 通过回车键来代替单击操作
driver.find_element_by_id('su').send_keys(Keys.ENTER)
```

运行上述代码就能看到键盘事件操作的过程。此外，Keys 类还定义了键盘上各个快捷键，具体的定义方式可以查看 Keys 类的源码，源码地址在 Python 安装目录的 Lib\site-packages\selenium\webdriver\common\keys.py 下。

9.5 常用功能

前几节中，我们已经学习了 Selenium 的基本使用方法，掌握了如何启动浏览器、查找并定位网页元素以及网页元素的操控。本节中，我们讲述 Selenium 的一些常用功能，如设置浏览器的参数、浏览器多窗口切换、设置等待时间、文件的上存与下载、Cookies 处理以及 frame 框架操作。

设置浏览器的参数是在定义 driver 的时候设置 chrome_options 参数，该参数是一个 Options 类所实例化的对象。其中常用的参数是设置浏览器是否可视化和浏览器的请求头等信息，前者可以加快代码运行速度，后者可以有效地防止网站的反爬虫检测。具体的代码如下：

```
from selenium import webdriver
# 导入 Options 类
from selenium.webdriver.chrome.options import Options
url = 'https://movie.douban.com/'
# Options 类实例化
chrome_options = Options()
# 设置浏览器参数
# --headless 是不显示浏览器启动以及执行过程
chrome_options.add_argument('--headless')
# 设置 lang 和 User-Agent 信息，防止反爬虫检测
chrome_options.add_argument('lang=zh_CN.UTF-8')
UserAgent='Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/68.0.3440.84 Safari/537.36'
chrome_options.add_argument('User-Agent=' + UserAgent)
# 启动浏览器并设置 chrome_options 参数
driver = webdriver.Chrome(chrome_options=chrome_options)
# 浏览器窗口最大化
# driver.maximize_window()
# 浏览器窗口最小化
# driver.minimize_window()
driver.get(url)
# 获取网页的标题内容
print(driver.title)
# page_source 是获取网页的 HTML 代码
print(driver.page_source)
```

浏览器多窗口切换是在同一个浏览器中切换不同的网页窗口。打开浏览器可以看到，浏览器顶部可以不断添加新的窗口，而 Selenium 可以通过窗口切换来获取不同的网页信息。具体代码如下：

```
from selenium import webdriver
import time
url = 'https://www.baidu.com/'
driver = webdriver.Chrome()
driver.get(url)
# 使用 JavaScript 开启新的窗口
js = 'window.open("https://www.sogou.com");'
driver.execute_script(js)
# 获取当前显示的窗口信息
current_window = driver.current_window_handle
# 获取浏览器的全部窗口信息
handles = driver.window_handles
# 设置延时可以看到切换效果
time.sleep(3)
# 根据窗口信息进行窗口切换
# 切换百度搜索的窗口
driver.switch_to_window(handles[0])
time.sleep(3)
# 切换搜狗搜索的窗口
driver.switch_to_window(handles[1])
```

上述代码中，使用了 execute_script 方法，这是通过浏览器运行 JavaScript 代码生成新的窗口，然后获取浏览器上的全部窗口信息，window_handles 方法是获取当前浏览器的窗口信息，并以列

表的形式表示，最后由 switch_to_window 方法进行窗口之间的切换。千万不要小看 execute_script 方法，很多浏览器的插件都是由 JavaScript 来实现的，可想而知它的作用是多么的强大。

Selenium 的执行速度相当快，在 Selenium 执行的过程中往往需要等待网页的响应才能执行下一个步骤，否则程序会抛出异常信息。网页响的应快慢取决于多方面因素，因此在某些操作之间需要设置一个等待时间，让 Selenium 与网页响应尽量达到同步执行，这样才能保证程序的稳健性。在前面的例子中，延时是使用 Python 内置的 time 模块来实现的，而 Selenium 本身提供了一些延时的功能，具体的使用方法如下：

```
    from selenium import webdriver
    url = 'https://www.baidu.com/'
    driver = webdriver.Chrome()
    driver.get(url)
    # 隐性等待，最长等待时间为 30 秒
    driver.implicitly_wait(30)
    driver.find_element_by_id('kw').send_keys('Python')
    # 显性等待
    from selenium.webdriver.support.wait import WebDriverWait
    from selenium.webdriver.common.by import By
    from selenium.webdriver.support import expected_conditions
    # visibility_of_element_located 检查网页元素是否可见
    # (By.ID, 'kw')：kw 是搜索框的 id 属性值，By.ID 是使用 find_element_by_id 定位
    condition = expected_conditions.visibility_of_element_located((By.ID, 'kw'))
    WebDriverWait(driver=driver, timeout=20,
poll_frequency=0.5).until(condition)
```

隐性等待是在一个设定的时间内检测网页是否加载完成，也就是一般情况下你看到浏览器标签栏那个小圈不再转，才会执行下一步。比如代码中设置 30 秒等待时间，网页只要在 30 秒内完成加载就会自动执行下一步，如果超出 30 秒就会抛出异常。值得注意的是，隐性等待对整个 driver 的周期都起作用，所以只要设置一次即可。

显性等待能够根据判断条件而进行灵活地等待，程序每隔一段时间检测一次，如果检测结果与条件成立了，则执行下一步，否则继续等待，直到超过设置的最长时间为止，然后抛出 TimeoutException 异常。显性等待的使用涉及到多个模块：By、expected_conditions 和 WebDriverWait。各个模块说明如下。

- By：设置元素定位方式。定位方式共 8 种，分别是 ID、XPATH、LINK_TEXT、PARTIAL_LINK_TEXT、NAME、TAG_NAME、CLASS_NAME、CSS_SELECTOR。
- expected_conditions：验证网页元素是否存在，提供了多种验证方式。具体可以查看源码：Lib\site-packages\selenium\webdriver\support\expected_conditions.py

WebDriverWait 的参数说明如下。

- driver：浏览器对象 driver。
- timeout：超时时间，等待的最长时间。
- poll_frequency：检测时间的间隔。
- ignored_exceptions：忽略的异常，如果在调用 until 或 until_not 的过程中抛出的异常在这个参数里，则不中断代码，继续等待，如果抛出的异常在这个参数之外，则中断代码并抛出

异常。默认值为 NoSuchElementException。

- until：条件判断，参数必须为 expected_conditions 对象。如果网页里某个元素与条件符合，则中断等待并执行下一个步骤。
- until_not：与 until 的逻辑相反。

隐性等待和显性等待相比于 time.sleep 这种强制等待更为灵活和智能，可解决各种网络延误的问题，隐性等待和显性等待可以同时使用，但最长的等待时间取决于两者之间的最大数，如上述代码的隐性等待时间为 30，显性等待时间为 20，则该代码的最长等待时间为隐性等待时间。

上存文件在网页中用上存按钮来显示，通过单击按钮就会打开本地电脑的一个文件对话框，在文件对话框选择文件并确认即可上存文件路径。而 Selenium 实现过程相对简单，只需定位到网页的上存按钮并使用 send_keys 方法来写入文件路径即可实现，如下所示：

```
# HTML 的元素信息
<div class="row-fluid">
<div class="span6 well">
<h3>upload file</h3>
<input type="file" name="file" />
</div>
</div>
# Selenium 定位
driver.find_element_by_name("file").send_keys("D:\file.txt")
```

在网页中，文件上存有多种实现方式，但无论哪一种方式，只要分析好上存的机制，都可以使用 Selenium 实现。而文件下载的原理与文件上存是一样的，具体代码如下：

```
from selenium import webdriver
# 设置文件保存的路径，如不设置，默认系统的 Downloads 文件夹
options = webdriver.ChromeOptions()
prefs = {'download.default_directory': 'd:\\'}
options.add_experimental_option('prefs', prefs)
# 启动浏览器
driver = webdriver.Chrome()
# 下载微信 PC 版安装包
driver.get('https://pc.weixin.qq.com/')
# 浏览器窗口最大化
driver.maximize_window()
# 单击下载按钮
driver.find_element_by_class_name('button').click()
```

下面讲述浏览器 Cookies 的使用，Cookies 操作无非就是读取、添加和删除 Cookies。Cookies 信息可以在浏览器开发者工具的 Network 标签页查看，查看步骤如图 9-10 所示。

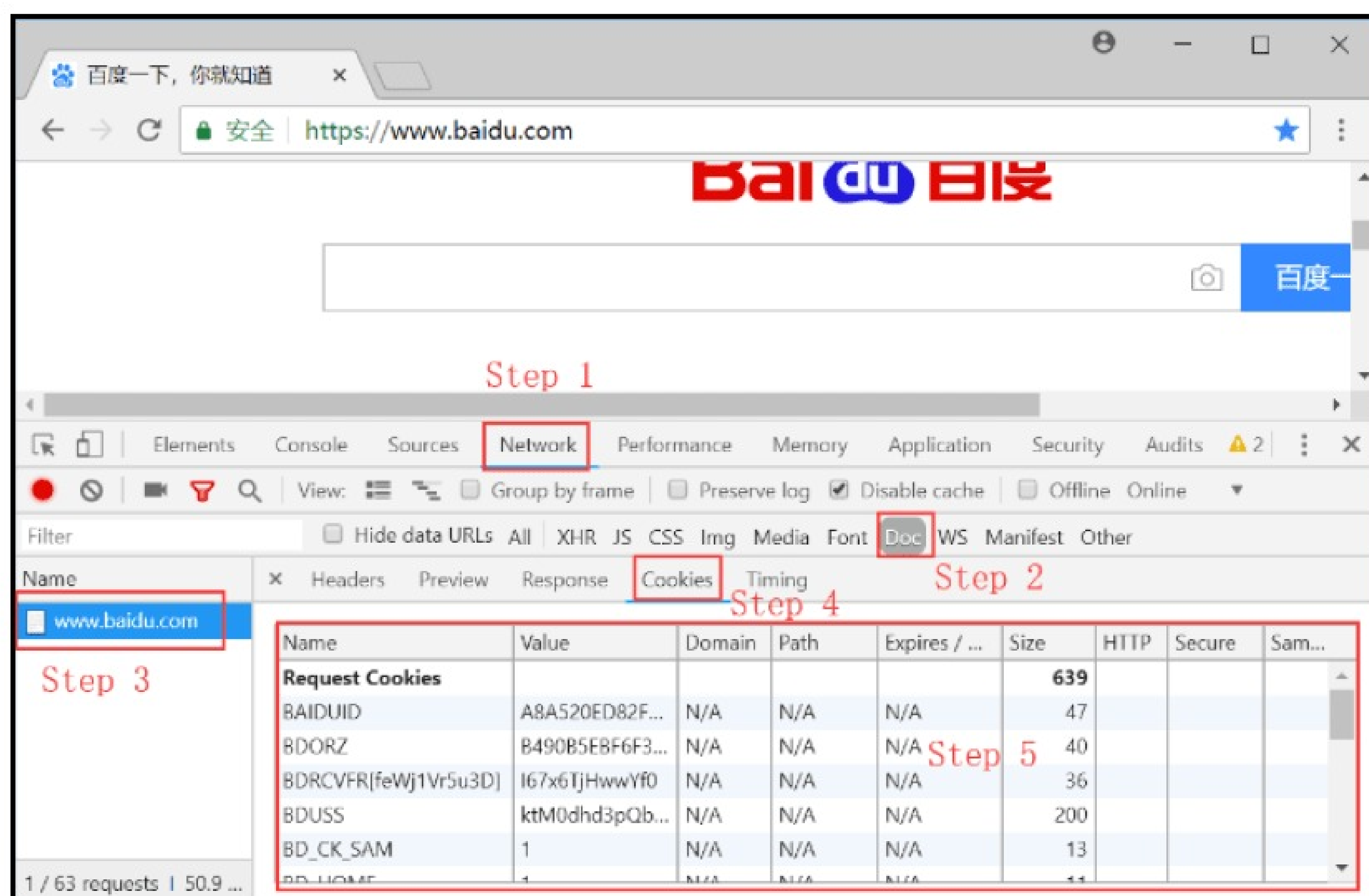


图 9-10　查看 Cookies 信息

从图 9-10 中可以看到，一个网页的 Cookies 可以有多条 Cookie 数据组成，每条数据都有 9 个属性。而我们需要检测 Selenium 获取 Cookies 信息与图上的数据格式是否一致，具体代码如下：

```
from selenium import webdriver
import time
# 启动浏览器
driver = webdriver.Chrome()
driver.get('https://www.youdao.com')
time.sleep(5)
# 添加 Cookies
driver.add_cookie({'name': 'Login_User', 'value': 'PassWord'})
# 获取全部 Cookies
all_cookies = driver.get_cookies()
print('全部的 Cookies 为：', all_cookies)
# 获取 name 为 Login_User 的 Cookie 内容
one_cookie = driver.get_cookie('Login_User')
print('单个的 Cookie 为：', one_cookie)
# 删除 name 为 Login_User 的 Cookie
driver.delete_cookie('Login_User')
surplus_cookies = driver.get_cookies()
print('剩余的 Cookie 为：', surplus_cookies)
# 删除全部 Cookies
driver.delete_all_cookies()
surplus_cookies = driver.get_cookies()
print('剩余的 Cookie 为：', surplus_cookies)
```

运行上述代码可以发现，代码输出的 Cookies 信息以列表的形式展示，列表的每个元素是一个字典，并且字典键值都与图上的 Cookies 信息一一对应。

frame 是一个框架页面，在 HTML5 已经不支持使用这个框架，但在一些网站中依然会看到它的身影。frame 的作用是在 HTML 代码里面嵌套一个或多个不同的 HTML 代码，每嵌套一个 HTML 都需要由 frame 来实现。以为百度知道的问题（https://zhidao.baidu.com/list?cid=110106）为例，打

开某条题目，题目的回答数最好是 0 回答，如图 9-11 所示。



图 9-11　百度知道问题列表

单击图 9-11 中的问题链接进入问题的详细信息页，并且打开开发者工具的 Elements 标签页，快速定位到文本输入框，在 Elements 标签页可以看到这个文本框是由 iframe 框架页面生成的。iframe 和 frame 实现的功能是相同的，只不过使用方式和灵活性有所不同，不管是 iframe 或 frame，Selenium 的定位和操作方式都是一样的。iframe 框架信息如图 9-12 所示。

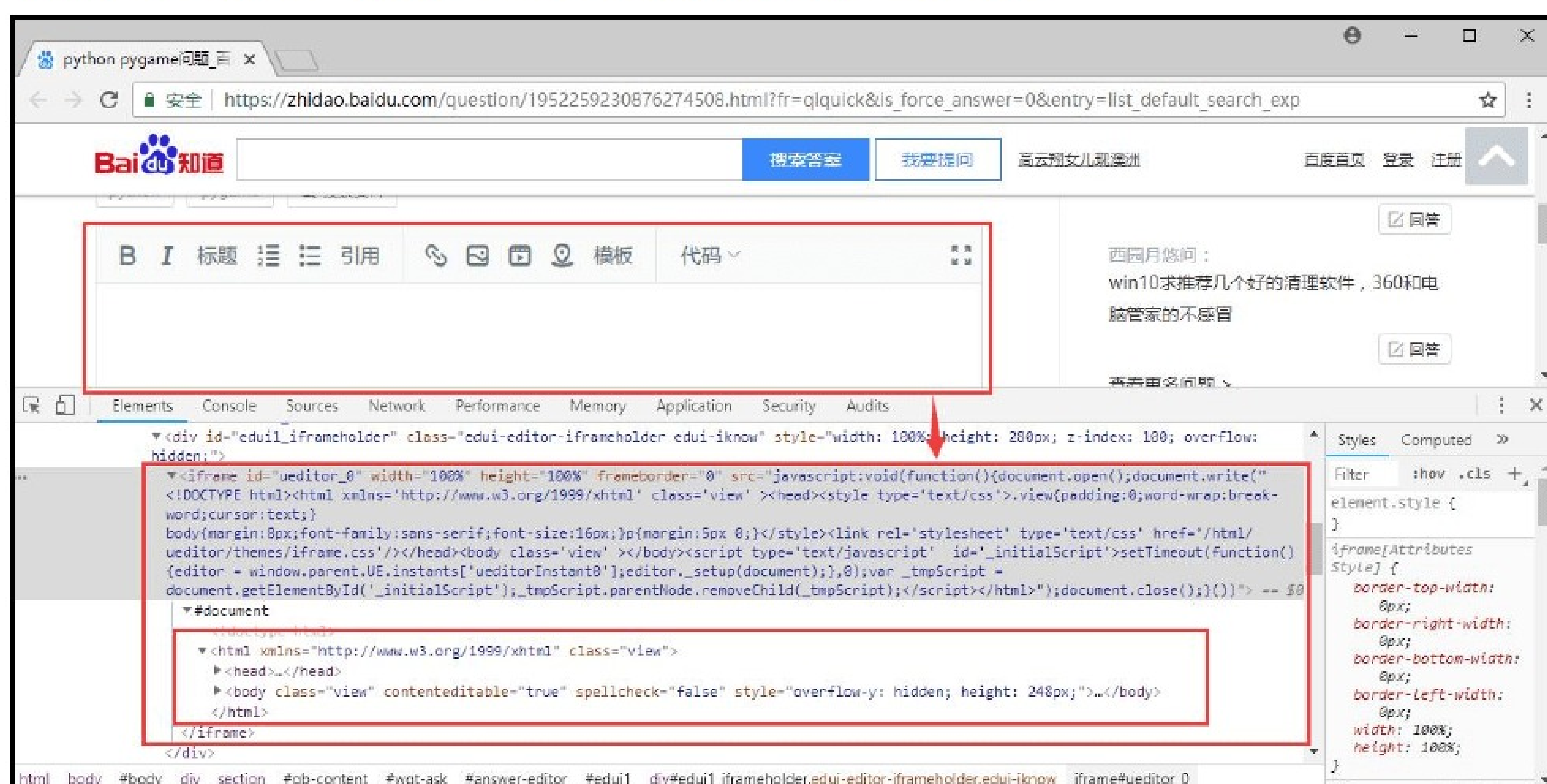


图 9-12　百度知道问题详细页

由于一个 HTML 可以嵌套了一个或多个 iframe，那么 Selenium 在操作不同的 iframe 需要通过 switch_to.frame()来切换到指定的 iframe，再执行相应的操作。比如一个网页中有多个 iframe，各个 iframe 的信息如图 9-13 所示。

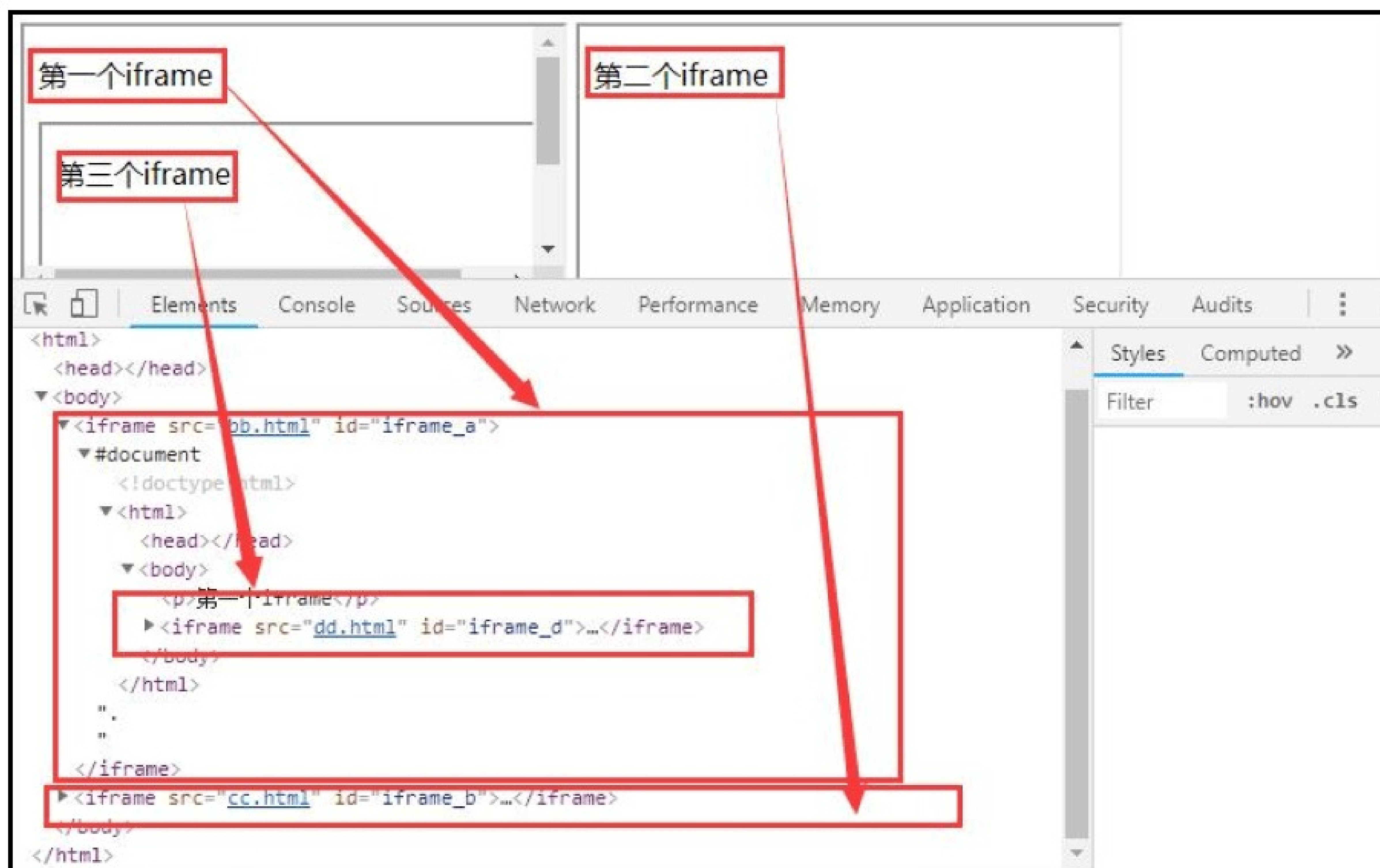


图 9-13　iframe 信息

图 9-13 中一共有 3 个 iframe，在当前网页里嵌套了 2 个 iframe，其中第一个 iframe 里面又嵌套了一个 iframe，不管是 HTML 切换 iframe，还是 iframe 之间的切换，实现过程都是由 switch_to 方法来完成。Selenium 对各个 iframe 的定位方法如下所示：

```
from selenium import webdriver
url = 'XXXXX'
driver = webdriver.Chrome()
driver.get(url)

""" 定位到第一个 iframe """
# 通过索引定位
driver.switch to.frame(0)
# 通过 iframe 的 id 或 name 属性定位
driver.switch_to.frame('iframe_a')
# 先定位 iframe 再切换到 iframe_a
element = driver.find element by id("iframe a")
driver.switch to.frame(element)
# 从 iframe a 跳回 HTML
driver.switch_to.default_content()

""" 定位到第二个 iframe """
# 通过索引定位
driver.switch to.frame(1)
# 通过 iframe 的 id 或 name 属性定位
driver.switch to.frame('iframe b')
# 先定位 iframe 再切换到 iframe b
element = driver.find element by id("iframe b")
driver.switch to.frame(element)
# 从 iframe_b 跳回 HTML
driver.switch to.default content()

""" 定位到第三个 iframe """
```

```
    # 定位到 iframe_a
    driver.switch_to.frame('iframe_a')
    # 再从 iframe_a 切换 iframe_d
    driver.switch_to.frame('iframe_d')
    # 从 iframe_d 跳回到 iframe_a
    driver.switch_to.parent_frame()
    # 从 iframe_d 跳回 HTML
    driver.switch_to.default_content()
```

9.6 实战：百度自动答题

本节通过使用 Selenium 来实现百度知道自动答题，在讲述之前，首先注册一个百度账号，在浏览器上打开 https://passport.baidu.com/v2/，使用手机号码即可完成注册，具体的注册过程不再详细讲述。

完成用户注册后，在浏览器上访问 https://zhidao.baidu.com/list?cid=110，该网页是显示某个分类的问题列表，每条问题代表一条链接，单击链接可以进入问题详情页。

在问题详情页里面，我们需要根据题目去搜索相关的答案，然后将答案写到问题详情页的回答文本框里，最后单击提交回答按钮即可实现答题。这个看似简单的功能却涉及到三个网页的操控。首先获取问题详情页的题目，然后根据题目搜索答案，在答案列表页中逐一访问每个答案的链接，在答案详情页中获取合理的答案，最后将答案写回到问题详情页中。整个过程如图 9-14 所示。

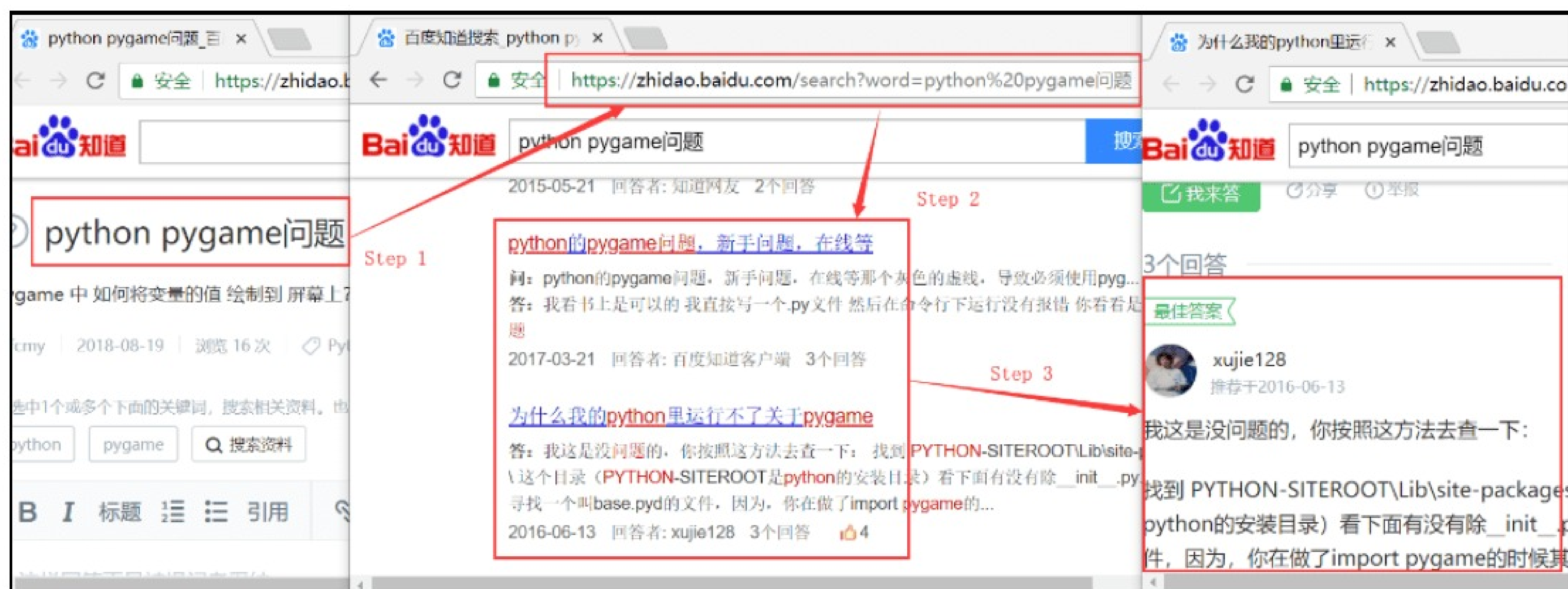


图 9-14 根据题目搜索答案

根据上述的简单分析，整个实战项目可以分为 5 个步骤来实现，每个步骤具体说明如下：

（1）在 https://zhidao.baidu.com/list?cid=110 上获取问题列表，得到全部问题的地址链接，然后遍历访问这些链接，依次进入问题的详情页。

（2）在问题详情页获取问题题目，题目是用于搜索相关的答案。

（3）搜索答案的地址链接都是固定的，如图 9-14 上所示，只要替换地址中 word 后面的内容即可搜索相关的答案。

（4）得到搜索结果后，获取答案列表的地址并遍历访问即可进入答案详情页，如果答案详情

页里面有最佳答案就会获取答案内容，并且终止答案列表的遍历。

（5）将得到的答案写回到问题详情页的回答文本框并单击提交回答按钮即可完成答题。

整个项目在实现过程中是在用户已登录的情况下执行，如果使用百度的账号密码去执行用户登录，就会遇到手机验证码或图片验证码。用户登录后，网站会一直保持用户的登录状态，不管用户是否重启浏览器，只要访问百度网址，用户登录信息就会显示出来。利用用户登录的状态，Selenium 可以模拟用户登录并将用户登录后的 Cookies 保存下来，在下次登录的时候，直接读取并操控 Cookies 即可完成用户登录。功能代码如下：

```
from selenium import webdriver
import json, time
# 百度用户登录并保存登录 Cookies
driver = webdriver.Chrome()
driver.get("https://www.baidu.com/")
driver.find_element_by_xpath('//*[@id="u1"]/a[7]').click()
time.sleep(3)
driver.find_element_by_id('TANGRAM__PSP_10__footerULoginBtn').click()
time.sleep(3)
# 设置用户的账号和密码
driver.find_element_by_xpath('//*[@id="TANGRAM__PSP_10__userName"]').
        send_keys('XX')
driver.find_element_by_xpath('//*[@id="TANGRAM__PSP_10__password"]').
        send_keys('XX')
try:
   verifyCode = driver.find_element_by_name('verifyCode')
   code_number = input('请输入图片验证码：')
   verifyCode.send_keys(str(code_number))
except: pass
driver.find_element_by_xpath('//*[@id="TANGRAM__PSP_10__submit"]').click()
time.sleep(3)
try:
    driver.find_element_by_xpath('//*[@id="TANGRAM__36__button_send_mobile"
        ]').click()
   code_photo = input('请输入短信验证码：')
   driver.find_element_by_xpath('//*[@id="TANGRAM__36__input_vcode"]').
        send_keys(str(code_photo))
   driver.find_element_by_xpath('//*[@id="TANGRAM__36__button_submit"]'
        ).click()
   time.sleep(3)
except: pass
cookies = driver.get_cookies()
f1 = open('cookie.txt', 'w')
f1.write(json.dumps(cookies))
f1.close()
```

上述代码使用了两次异常捕捉，这是用于检测图片验证码和短信验证码是否存在，两种验证方式是否出现取决于百度账号的安全性设置以及网络环境等因素。每个操作之间都设置了强制性延时，这是为了让程序与网页之间能够同步协调。最后完成整个登录操作后，将网页的 Cookies 信息保存到 txt 文件中。

得到用户的登录信息，接下来实现自动答题。整个答题过程一共涉及 4 个网页：百度知道问题列表页、百度知道问题详情页、答案搜索页和答案详情页。

在问题列表页中，每条问题的 HTML 代码是由标签<a>生成，并且属性 class 的属性值为 `title-link`，如图 9-15 所示。因此 Selenium 可以对属性 class 进行定位，获取全部问题所在的标签<a>，遍历这些标签提取相应的链接地址。

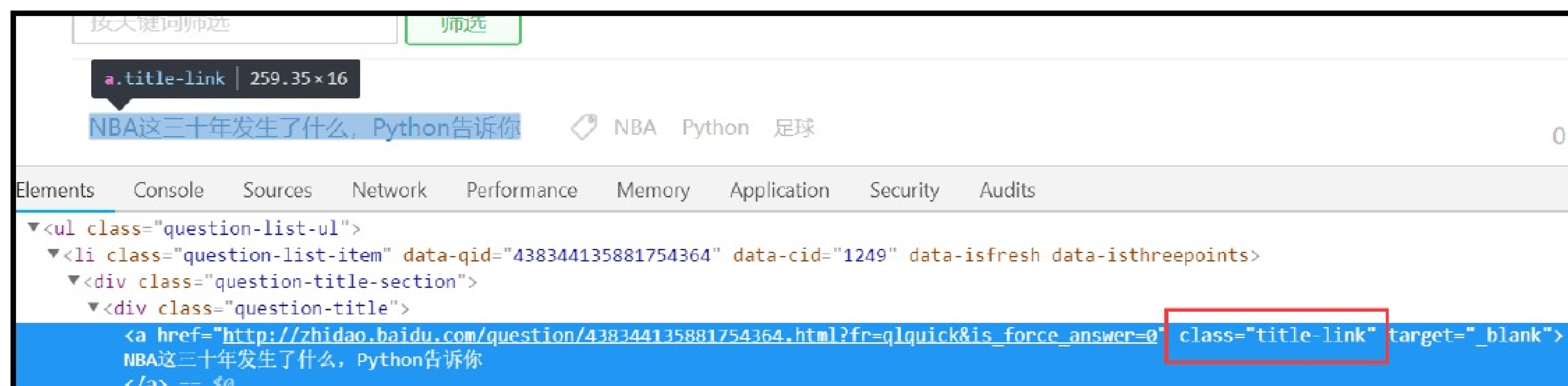


图 9-15　问题列表页

在新的窗口访问每条问题链接，这些链接会进入相应的问题详情页。在问题详情页中，首先判断问题是否已被抢答，如果尚未被回答，程序根据题目去百度知道搜索相关的答案，在这些相关答案中找到最佳答案，然后写入问题答案输入框里并单击“提交回答”按钮；如果问题已被回答，程序就关闭当前窗口，回到问题列表执行下一个问题。问题详情页的答案输入框和“提交回答”按钮的 HTML 代码如图 9-16 所示。

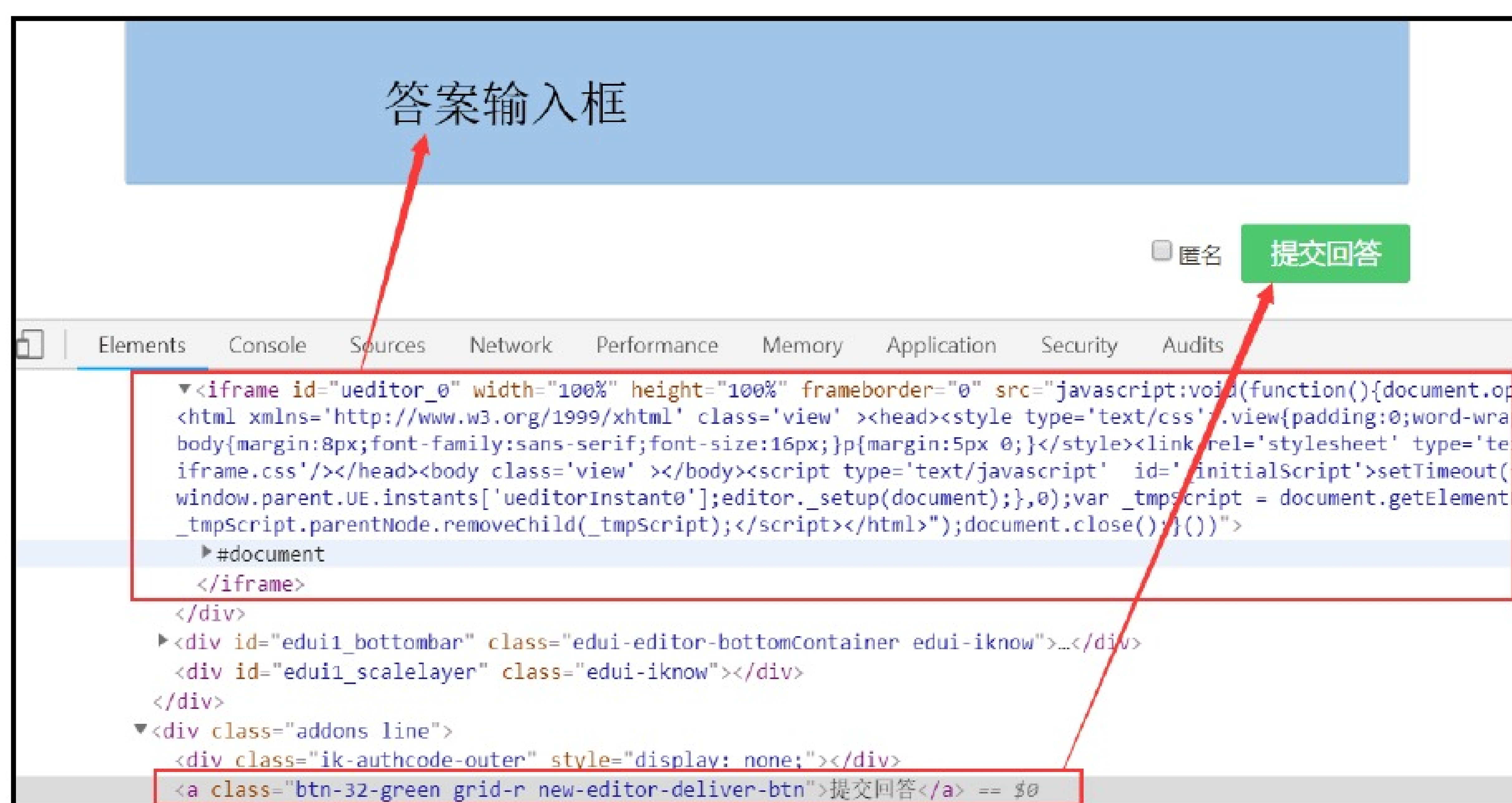


图 9-16　问题详情页

回答问题的过程中涉及到两个新的网页：答案搜索页和答案详情页。答案搜索页是根据问题在新的窗口中搜索相关答案，每个答案的链接以标签<dt>表示，该标签下含有标签<a>。将 Selenium 定位到每个答案的标签<a>，再获取 href 属性值，该属性值用于进入答案详情页，如图 9-17 所示。

将答案详情页的链接在新的窗口里访问，每个答案详情页都不一定有最佳答案，根据分析可知，最佳答案的 class 属性值为 best-text mb-10，如果 Selenium 能对属性 class 进行定位，则说明当前答案详情页有最佳答案，反之则无，如图 9-18 所示。

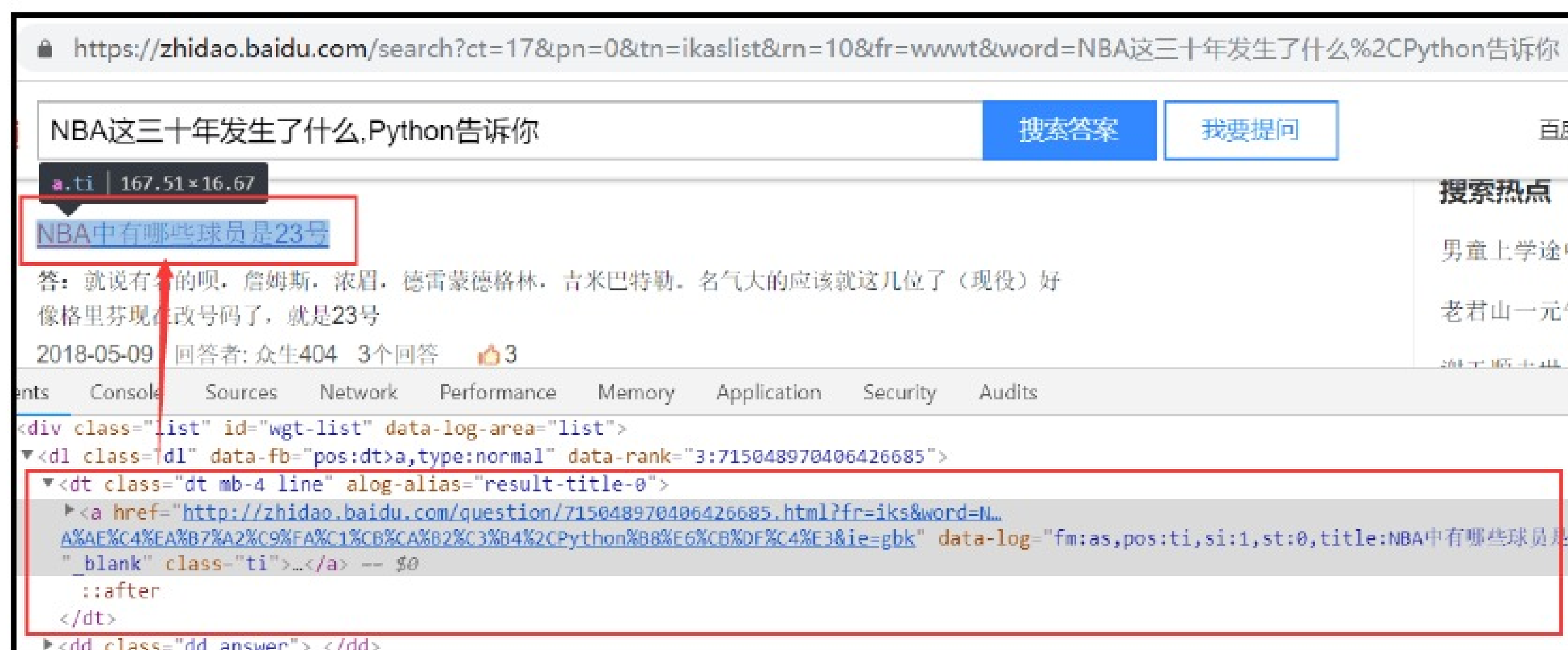


图 9-17　答案搜索页

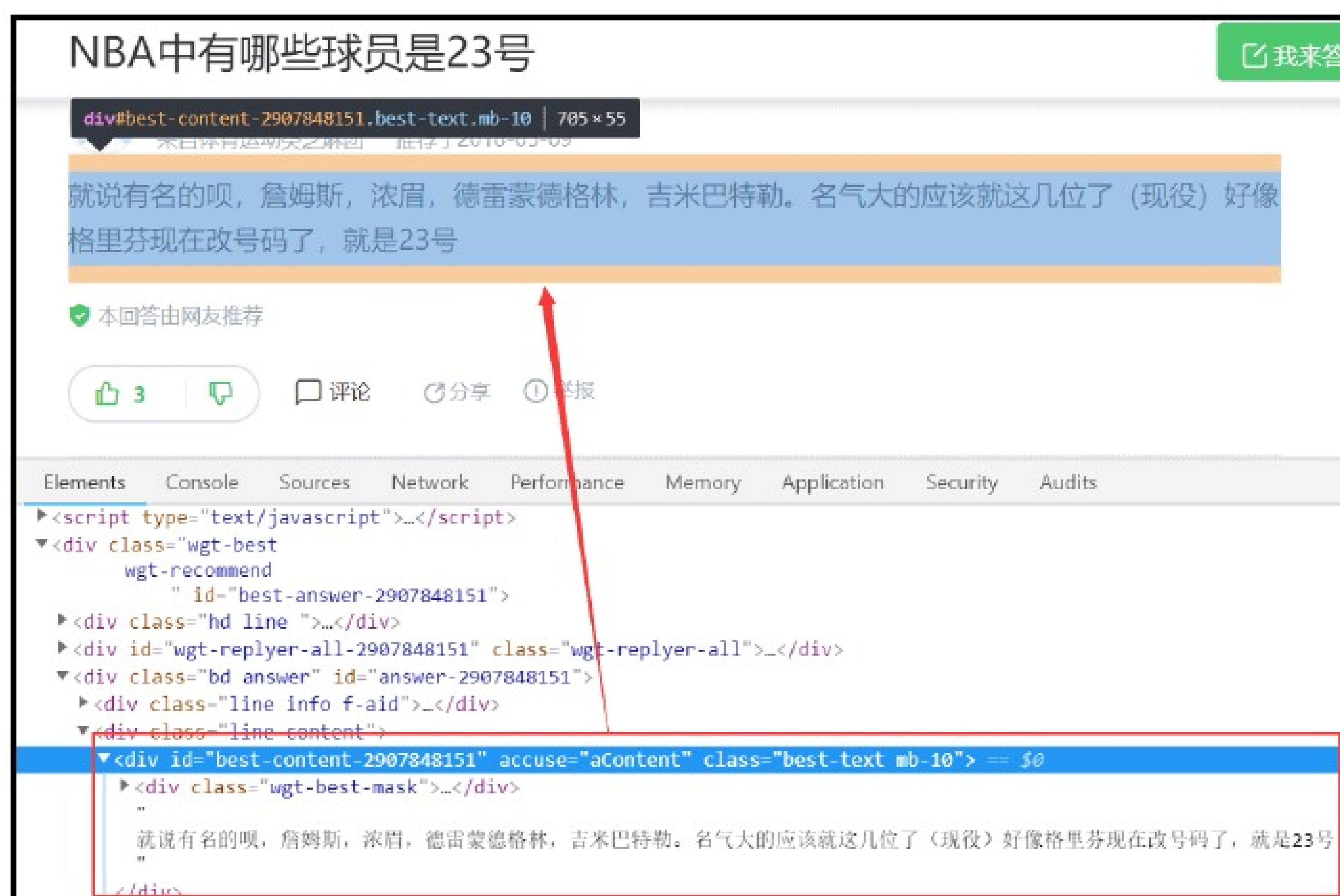


图 9-18　答案详情页

根据上述的元素定位以及答题的业务逻辑，整个答题程序需要注意每个页面窗口之间的切换，如果窗口的切换逻辑不严谨，很容易导致程序出错。此外还需要考虑一些异常的情况出现，比如问题搜不到任何答案、问题已被回答以及网络延时响应等一些特殊情况。综合分析，自动答题的功能代码如下所示：

```
from selenium import webdriver
import json, time
url = 'https://zhidao.baidu.com/list?cid=110'
driver = webdriver.Chrome()
driver.get(url)
# 使用 Cookies 登录
driver.delete_all_cookies()
f1 = open('cookie.txt')
cookie =json.loads(f1.read())
f1.close()
for c in cookie:
    driver.add_cookie(c)
driver.refresh()

# 获取问题列表
title_link = driver.find_elements_by_class_name('title-link')
```

```
    for i in title_link:
        # 打开问题详细页并切换窗口
        driver.switch_to.window(driver.window_handles[0])
        href = i.get_attribute('href')
        driver.execute_script('window.open("%s");' % (href))
        time.sleep(5)
        driver.switch_to.window(driver.window_handles[1])
        try:
            # 查找 iframe，判断问题是否已被回答
            driver.find_element_by_id('ueditor_0')
            # 获取问题题目并搜索答案
            title = driver.find_element_by_class_name('ask-title ').text
            title_url = 'https://zhidao.baidu.com/search?&word=' + title
            js = 'window.open("%s");' % (title_url)
            driver.execute_script(js)
            time.sleep(5)
            driver.switch_to.window(driver.window_handles[2])
            # 获取答案列表
            answer_list = driver.find_elements_by_class_name('dt,mb-4,line')
            for k in answer_list:
                # 打开答案详细页
                href = k.find_element_by_tag_name('a').get_attribute('href')
                driver.execute_script('window.open("%s");' % (href))
                time.sleep(5)
                driver.switch_to.window(driver.window_handles[3])
                # 获取最佳答案
                try:
                    text = driver.find_element_by_class_name('
                            best-text,mb-10'). text
                except:
                    text = ''
                finally:
                    # 关闭答案详情页的窗口
                    driver.close()
                # 答案不为空
                if text:
                    # 关闭答案列表页的窗口
                    driver.switch_to.window(driver.window_handles[2])
                    driver.close()
                    # 将答案写在问题回答文本框上并单击提交答案按钮
                    driver.switch_to.window(driver.window_handles[1])
                    driver.switch_to.frame('ueditor_0')
                    driver.find_element_by_xpath('/html/body').click()
                    driver.find_element_by_xpath('/html/body').send_keys(text)
                    # 跳回到网页的 HTML
                    driver.switch_to.default_content()
                    # 单击提交回答按钮
                    driver.find_element_by_xpath('//*[@id="answer-editor"]/
                            div[2]/a').click()
                    time.sleep(5)
                    # 关闭问题详细页的窗口
                    driver.switch_to.window(driver.window_handles[1])
                    driver.close()
                    break
```

```
        except Exception as err:
            # 除了问题列表页，关闭其他窗口
            all_handles = driver.window_handles
            for i, v in enumerate(all_handles):
                if i != 0:
                    driver.switch_to.window(v)
                    driver.close()
            driver.switch_to.window(driver.window_handles[0])
            print(err)
```

上述代码多次使用了 try…except 异常机制，这是为处理一些特殊情况，在一定程度上保证了程序的稳健性。程序中涉及到 4 个网页都是使用 JavaScript 去打开新的窗口，使用 JavaScript 也是为了提高程序的稳健性，因为 Selenium 的 click()方法没有 JavaScript 稳定，读者不妨将 JavaScript 的代码改用 click()方法去实现，测试程序的稳定性，就会发现效果不同。

9.7　本 章 小 结

Selenium 是一个用于网站应用程序自动化的工具，它可以直接运行在浏览器中，就像真正的用户在操作一样。它支持的浏览器包括 IE、Mozilla Firefox、Safari、Google Chrome 和 Opera 等，同时支持多种编程语言，如.Net、Java、Python 和 Ruby 等。

搭建 Selenium 开发环境需要安装 Selenium 库并且配置 Google Chrome 的 WebDriver。安装 Selenium 库可以使用 pip 指令完成；配置 Google Chrome 的 WebDriver 首先通过浏览器版本确认 WebDriver 的版本，然后下载相应的 WebDriver 并存放在 Python 的安装目录。

Selenium 定位网页元素主要通过元素的属性值或者元素在 HTML 里的路径位置，定位方式有以下 8 种：

```
# 通过属性 id 和 name 来实现定位
find_element_by_id()
find_element_by_name()

# 通过 HTML 标签类型和属性 class 实现定位
find_element_by_class_name()
find_element_by_tag_name()

# 通过标签值实现定位，partial_link 用于模糊匹配。
find_element_by_link_text()
find_element_by_partial_link_text()

# 元素的路径定位选择器
find_element_by_xpath()
find_element_by_css_selector()
```

Selenium 可以模拟任何操作，比如单击、右击、拖拉、滚动、复制粘贴或者文本输入等。操作方式分为三大类：常规操作、鼠标事件操作和键盘事件操作。

Selenium 还有一些常用功能，如设置浏览器的参数、浏览器多窗口切换、设置等待时间、文件的上存与下载、Cookies 处理以及 frame 框架操作。

第 10 章

手机 App 数据爬取

10.1　Appium 简介及原理

Appium 是一个开源、跨平台的测试框架，可以用来测试原生及混合的移动端应用。Appium 支持 iOS、Android 及 FirefoxOS 平台，它使用 WebDriver 的 JSON Wire 协议来驱动 iOS 系统的 UIAutomation 库以及 Android 系统的 UIAutomator 框架。它允许自动化人员在不同的平台（iOS 和 Android）使用同一套 API 来写自动化脚本，这样大大增加了 iOS 和 Android 的代码复用性。

整个 Appium 分为 Client 和 Server 两部分：Client 封装了 Selenium 客户端类库，为用户提供所有常见的 Selenium 命令以及额外的移动设备控制相关的命令，如多点触控手势和屏幕朝向等；Server 定义了官方协议的扩展，为用户提供了方便的接口来执行各种设备的行为，例如在测试过程中安装/卸载 App 等。

Appium 支持多种编程语言开发自动化程序，这取决于它选择了 Client/Server 的设计模式。Client 通过发送 HTTP 请求给 Server，当 Server 接收到 Client 发送的请求，解析请求内容并调用对应的系统框架，在移动设备上执行自动化操作。因为 Client 和 Server 之间是采用 HTTP 协议，所以 Client 用什么语言来开发自动化程序都是可以的。Appium 的工作原理如图 10-1 所示。

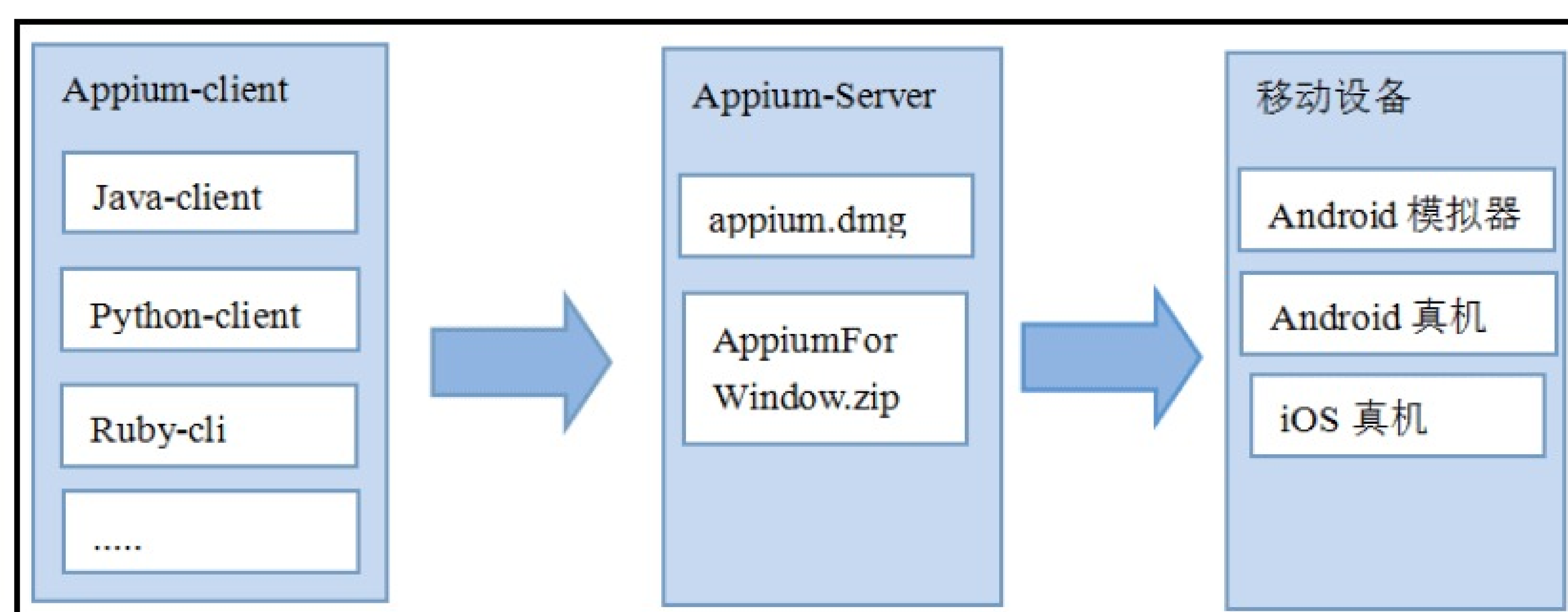


图 10-1　Appium 工作原理

从 Appium 的原理图可以看到，Appium-Client 能为我们提供自动化功能模块，用于编写自动化程序。在 Python 中，它是第三方模块 Appium，该模块是在 Selenium 库的基础上进行封装。Appium-Server 是基于 Node.JS 开发的服务端，主要接收 Appium-Client 的请求，根据请求信息去操作移动设备，从而实现自动化操作。

10.2　搭建开发环境

Appium 支持 Android 和 iOS 系统的移动设备自动化开发，但是苹果设备的自动化程序必须在 Mac 下进行开发，Windows 和 Linux 平台是无法完成的，因此我们以 Android 系统为例。

在 Windows 系统上搭建 Appium 开发环境，需要安装 JavaJDK、Android SDK、Node.JS、Appium-Server 和 Appium-Client，具体的安装说明如下：

步骤01 Java JDK：搭建 Java 的开发环境。

步骤02 Android SDK：Android 软件开发包，基于 Java 的开发环境运行，可以在计算机启用 Android 模拟器或者连接 Android 手机。

步骤03 Node.JS：搭建 Node.JS 的开发环境。

步骤04 Appium-Server：安装 Appium 的服务器，基于 Node.JS 的开发环境运行。

步骤05 Appium-Client：安装 Appium 的客户端，编写并运行 Appium 自动化代码。

Java JDK 是在 Windows 上搭建 Java 的开发环境，因为 Android SDK 是基于 Java 的开发环境运行的。目前 Java 最新版本是 10.0，但 Android SDK 仅支持 Java 8 版本，因此我们需要安装 Java 8 版本，在浏览器中访问 http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html，下载与计算机系统匹配的安装包，如图 10-2 所示。

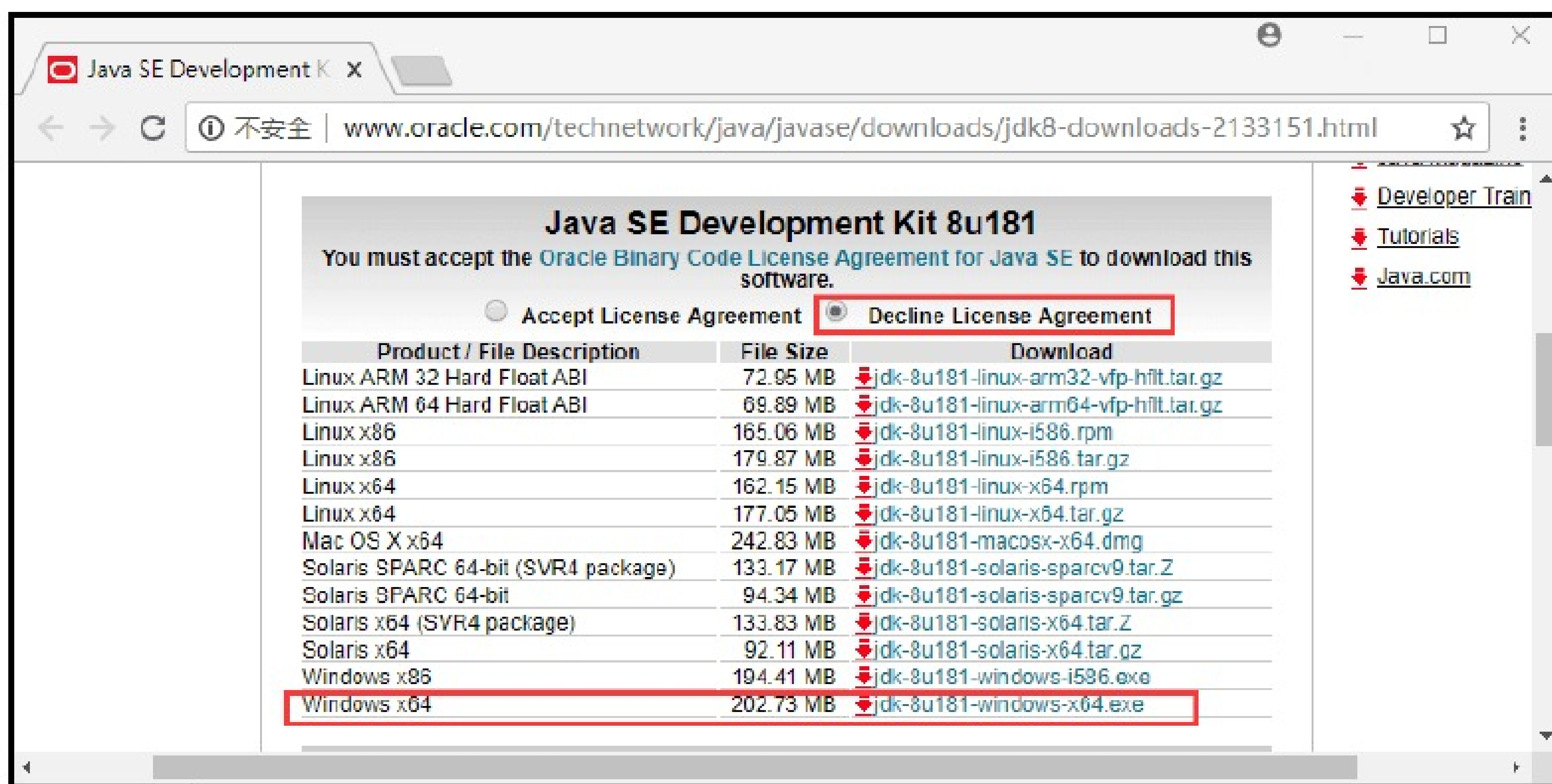


图 10-2　Java 版本下载

安装包下载后直接双击运行并根据安装提示即可完成安装，安装路径使用默认设置即可。安装成功后，还需要设置计算机的系统环境变量。右键单击我的电脑→选择属性→选择系统保护→选择高级→单击环境变量→单击系统变量的新建按钮，分别输入变量名 JAVA_HOME 和变量值 C:\Program Files\Java\jdk1.8.0_181，变量值是 Java 的默认安装路径，具体操作如图 10-3 所示。

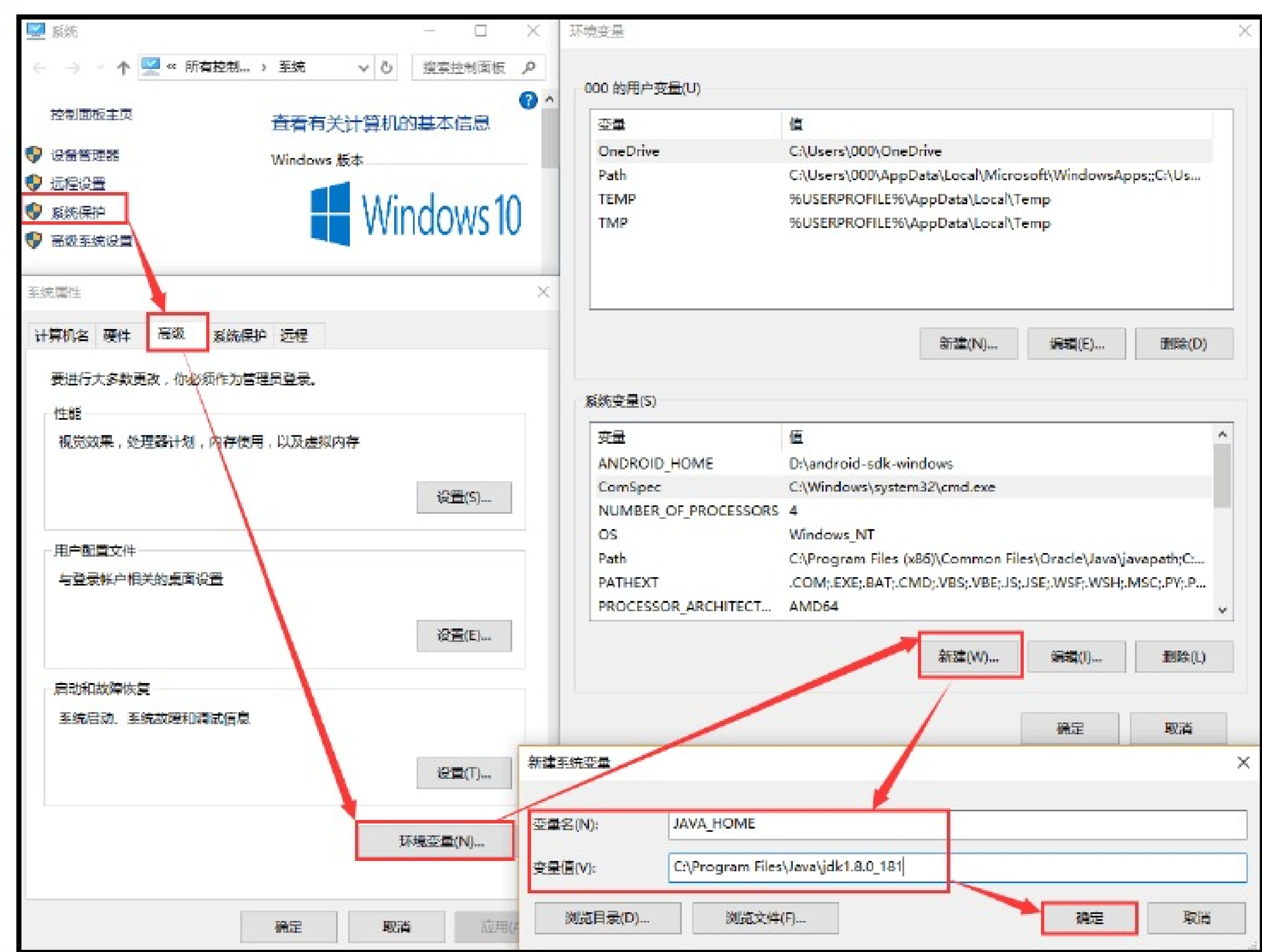


图 10-3　设置 Java 环境变量

Java 环境变量设置成功后，打开 CMD 窗口来验证 Java 是否安装成功。在 CMD 窗口输入 java -version 并按回车就会显示当前 Java 的版本信息，如图 10-4 所示。

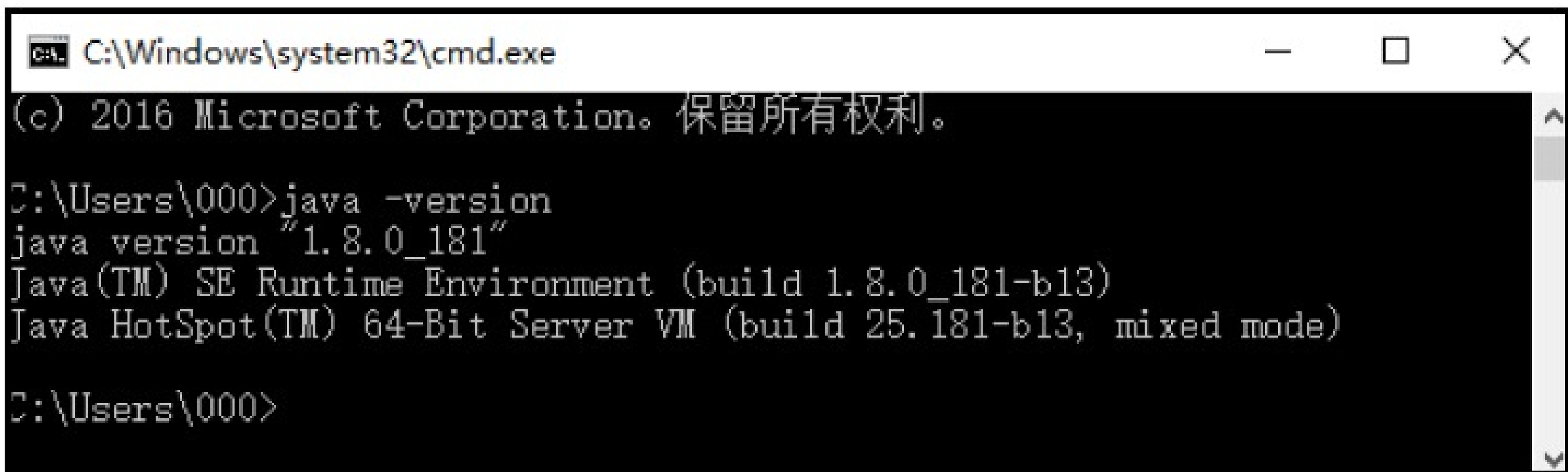


图 10-4　Java 版本信息

Java 的开发环境搭建后，下一步是搭建 Android SDK。Android SDK 提供了 Android API 库和开发工具构建、测试和调试应用程序。简单来讲，Android SDK 可以用于开发和运行 Android 系统的应用软件。在官网上没有找到单独的 Android SDK 下载链接，官方推荐下载包含 Android SDK 的 Android Studio。只能通过其他路径下载，在浏览器中访问 http://tools.android-studio.org/index.php/sdk，单击下载 android-sdk_r24.4.1-windows.zip，如图 10-5 所示。

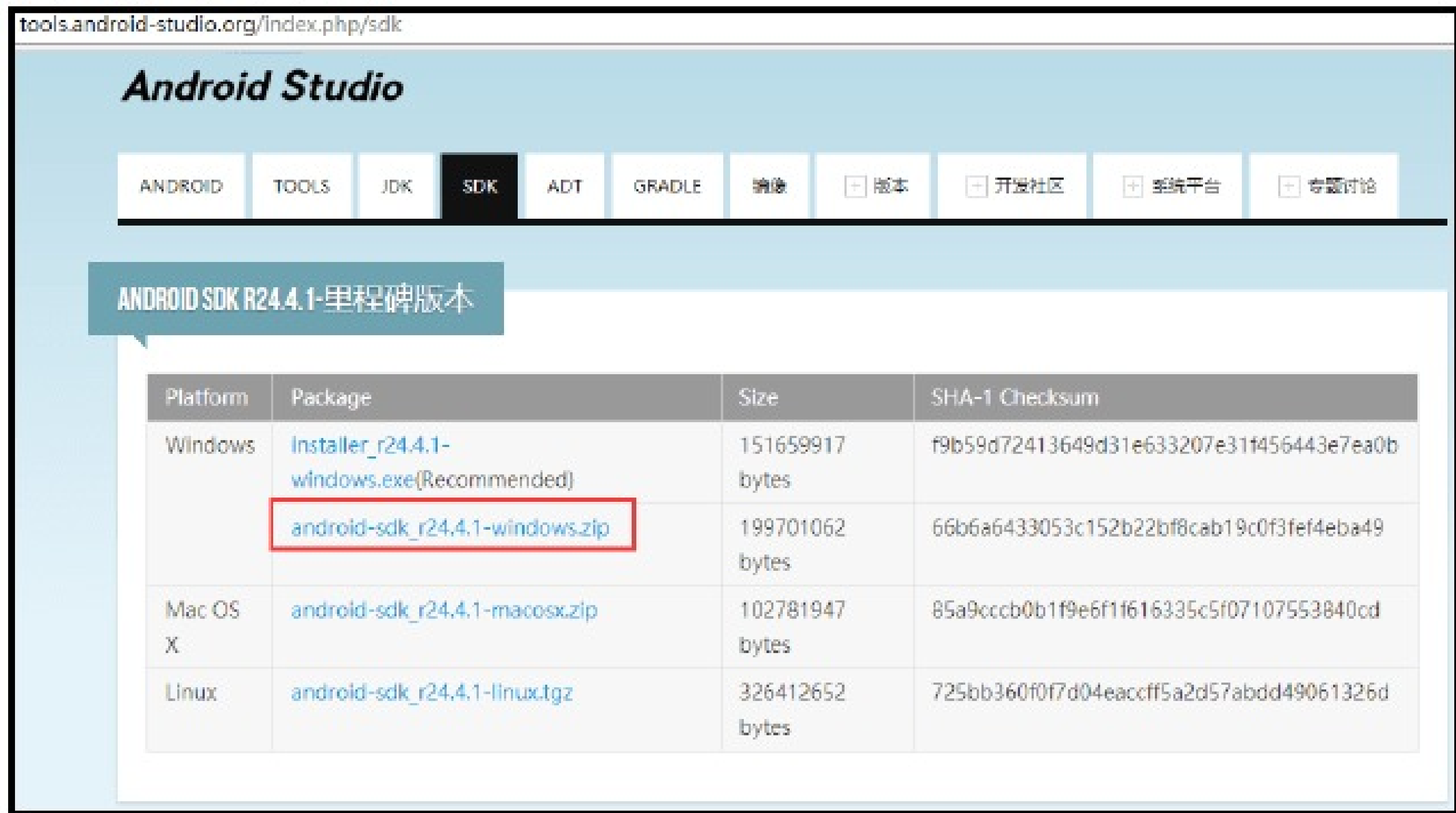


图 10-5　下载 Android SDK

将 android-sdk_r24.4.1-windows.zip 进行解压并放置在 D 盘的 SDK 文件夹里面，放置的路径没有具体要求，只要存在的空间足够大即可，因为后续更新 Android SDK 会占用比较大的存储空间。Android SDK 的路径信息如图 10-6 所示。

此电脑 › 软件 (D:) › SDK › android-sdk-windows

名称	修改日期	类型	大小
add-ons	2015/10/14 9:51	文件夹	
platforms	2015/10/14 9:51	文件夹	
tools	2018/9/17 16:53	文件夹	
AVD Manager.exe	2015/10/14 9:51	应用程序	216 KB
SDK Manager.exe	2015/10/14 9:51	应用程序	216 KB
SDK Readme.txt	2015/10/14 9:51	文本文档	2 KB

图 10-6　Android SDK 的文件信息

根据图 10-6 中的文件路径信息，将其添加到计算机的系统环境变量中，添加方式与 Java 的相似。新增变量 ANDROID_HOME，变量值是 Android SDK 的文件路径，如图 13-7 所示。在系统变量 Path 添加两个变量值，分别是 Android SDK 的 platform-tools 和 tools 文件夹的文件路径，如图 13-8 所示。

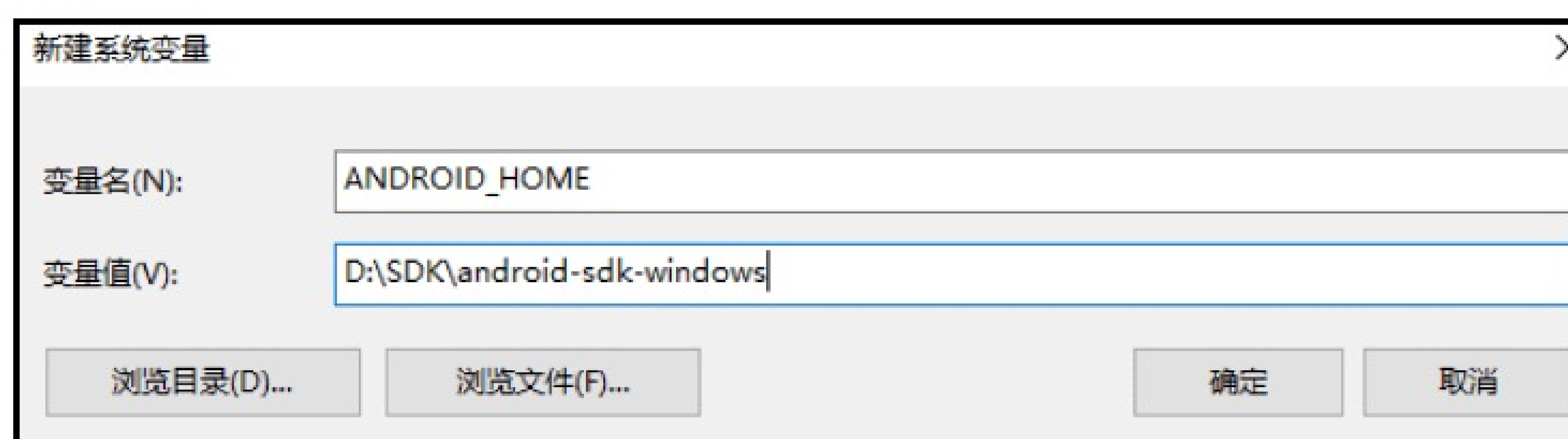


图 10-7　添加变量 ANDROID_HOME

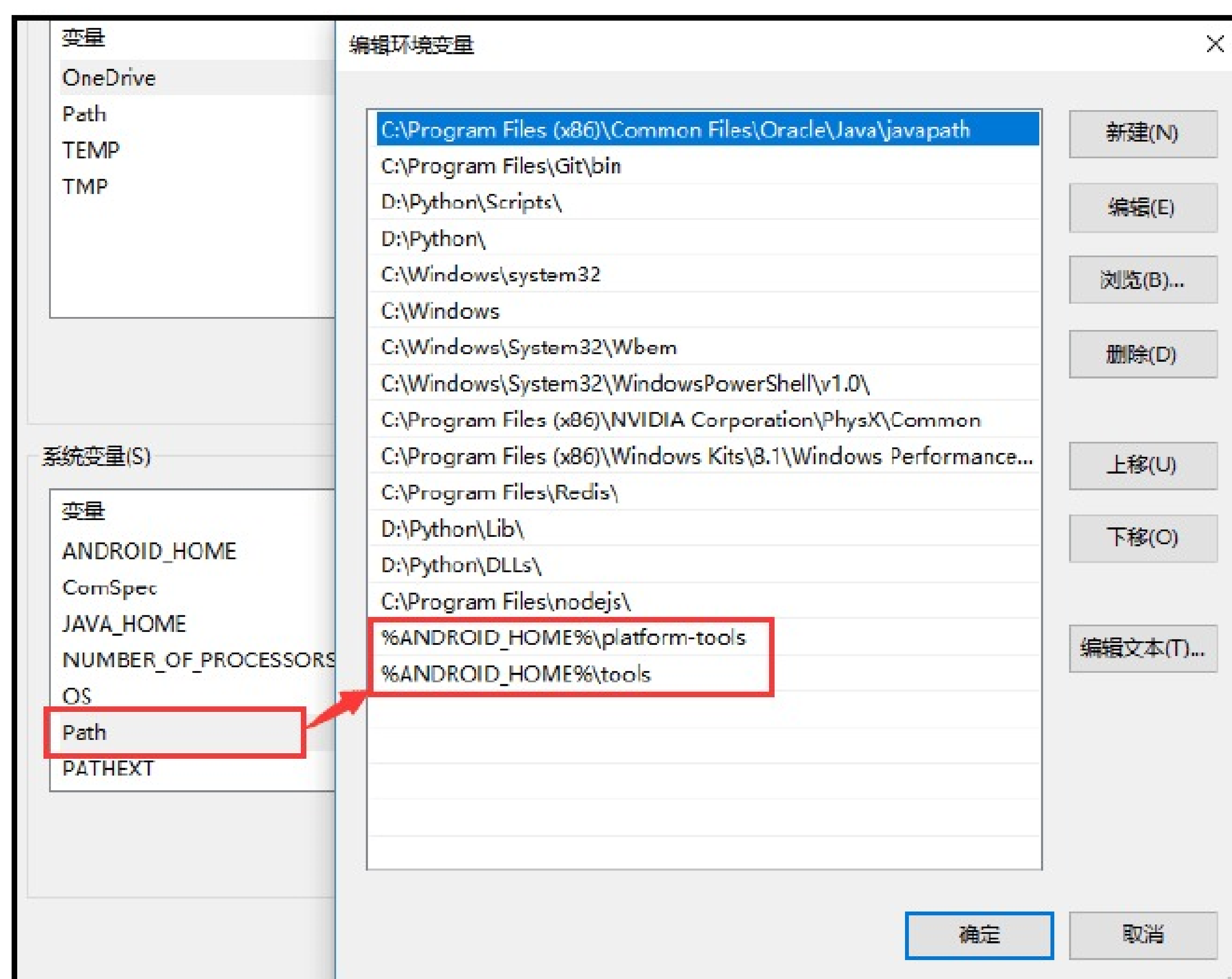


图 10-8　变量 Path 添加变量值

双击运行 SDK Manager.exe，这是更新安装 SDK 的版本信息。根据实际需求选择安装 Android 版本，比如本书的 Android 手机系统版本是 Android 8.0，Android 模拟器是 5.0 版本，安装选项如

图 10-9 所示。

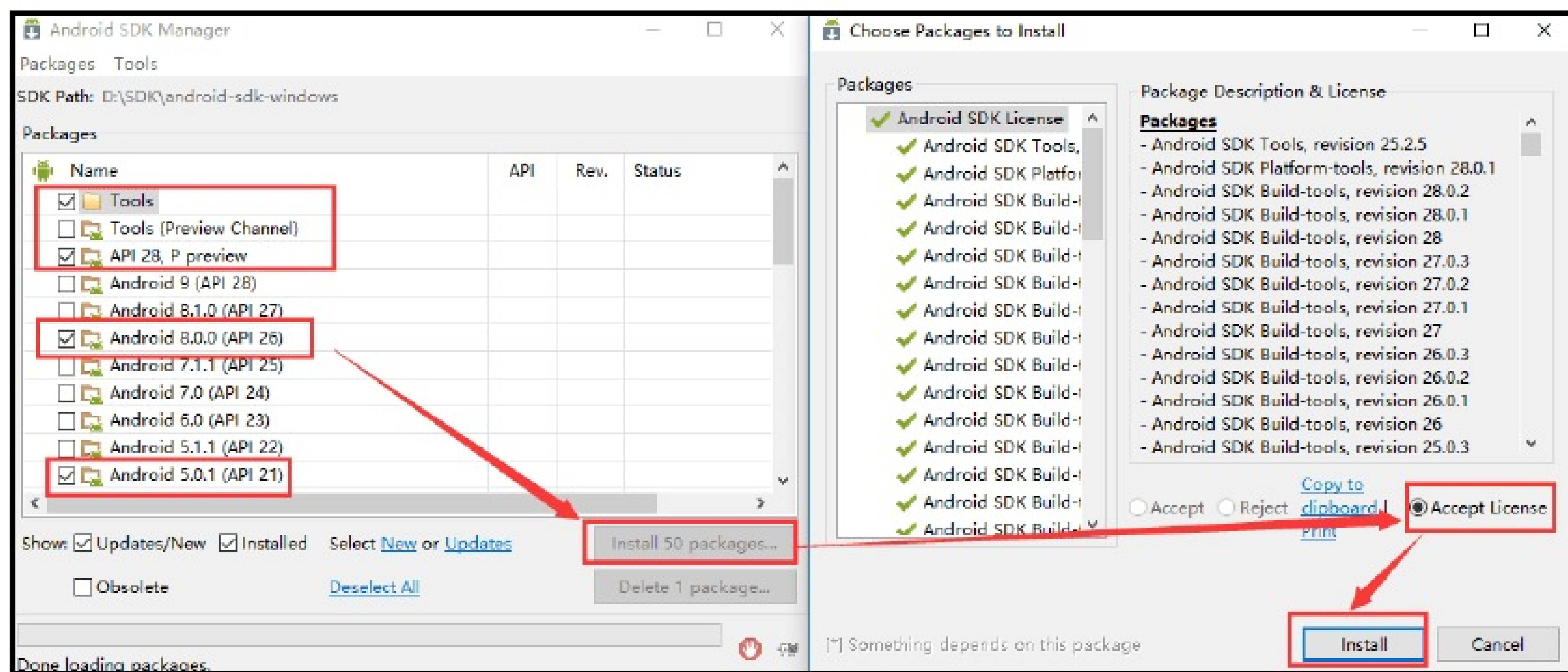


图 10-9 Android SDK 安装选项

完成 Android SDK 的更新后，打开 AVD Manager.exe 来创建 Android 模拟器。Android 模拟器是能在电脑上模拟 Android 操作系统，可以安装、使用、卸载 Android 应用的软件，它让你在电脑上也能体验操作 Android 系统的全过程。

在 AVD Manager 界面上单击“Create”按钮会出现 Android 模拟器的配置信息，填写配置信息后单击“OK”按钮就能创建 Android 模拟器，如图 10-10 所示。

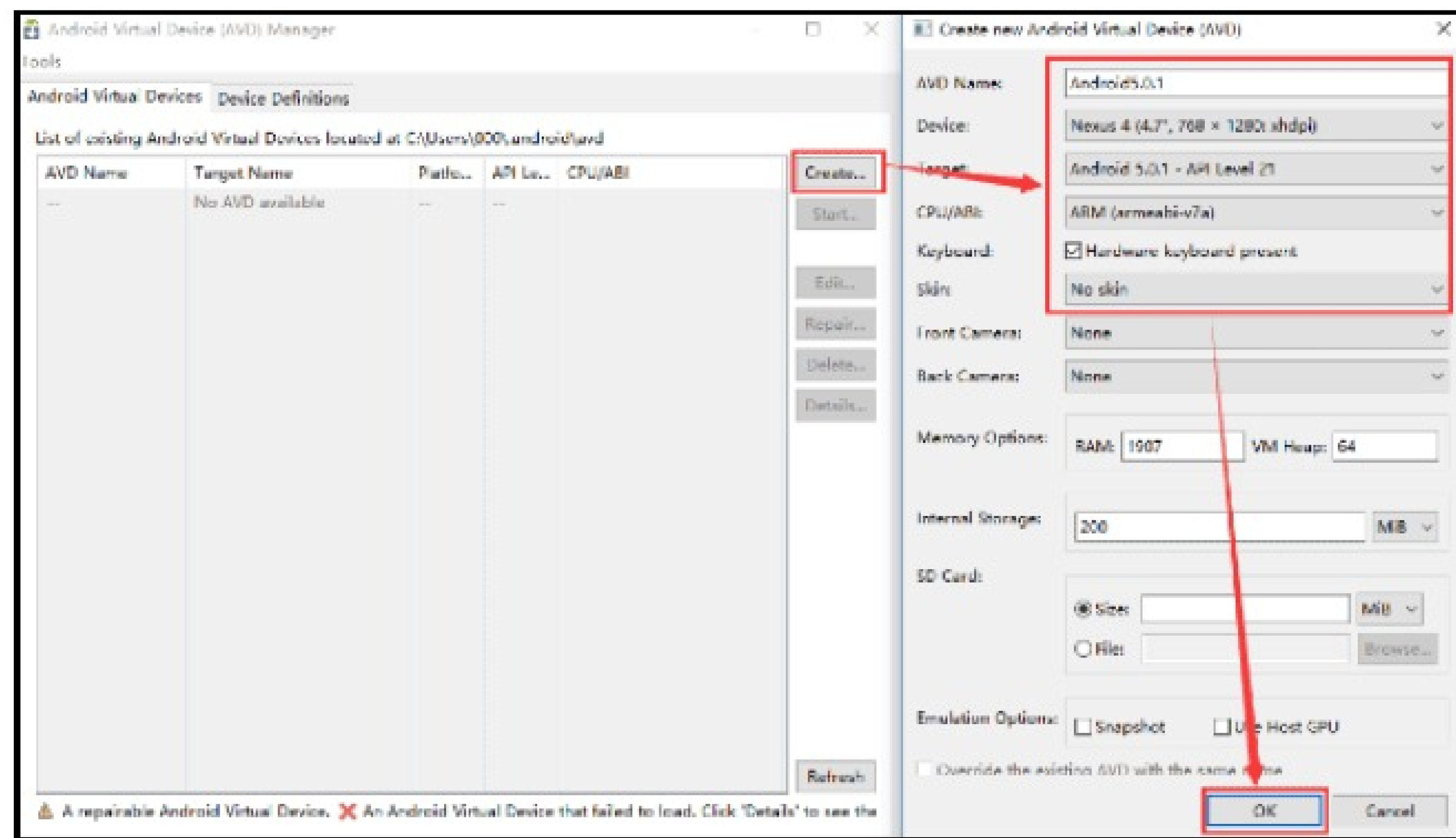


图 10-10 创建 Android 模拟器

Android 模拟器创建后，在 AVD Manager 界面可以看到刚创建的模拟器信息，使用鼠标选中模拟器信息并单击“Start”按钮→单击“Launch”按钮即可运行 Android 模拟器，Android 模拟器开启时间相对较长，需要耐心等待。如图 10-11 所示。

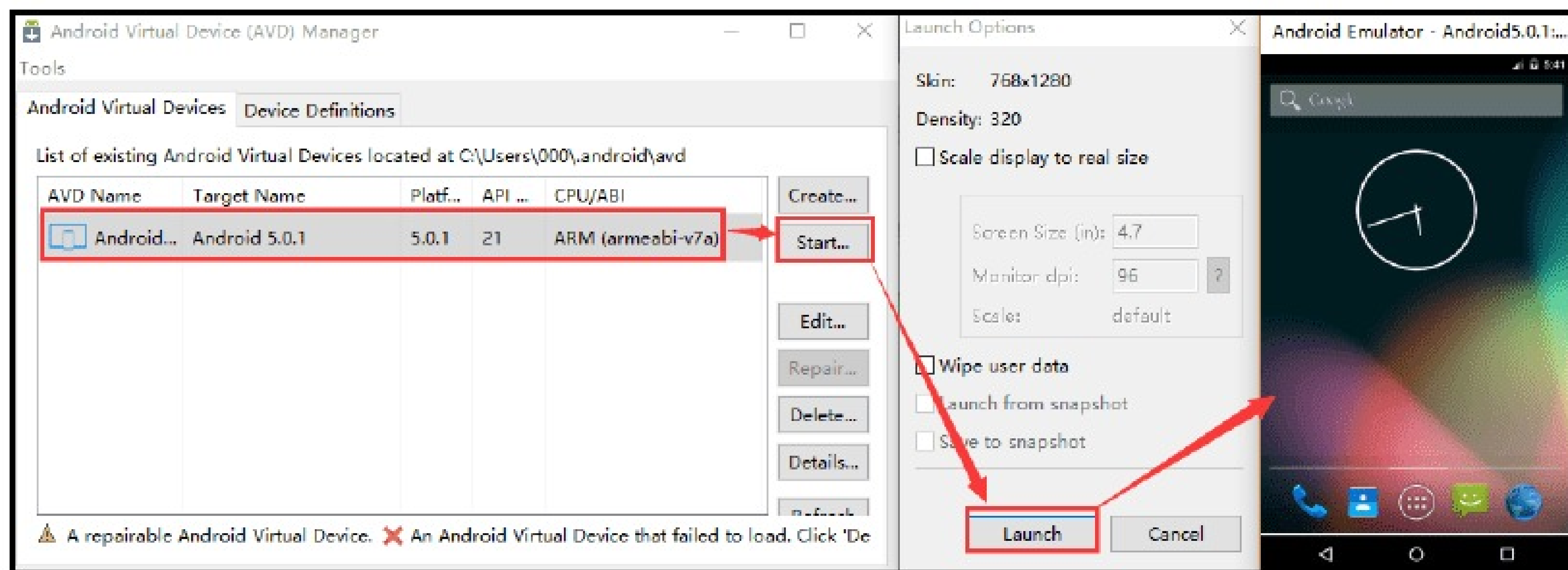


图 10-11 启动 Android 模拟器

最后测试 Android SDK 与手机的连接是否成功，手机通过 USB 连接电脑，并且开启手机的开发者模式以及安装相应的驱动程序。不同手机的开发者模式的开启方法各不相同，本书就不详细讲述，具体的开启方法可自行网上查询，手机的驱动程序安装成功后可以在设备管理器查看。完成上述操作后，在 CMD 窗口输入指令 adb devices 查看手机信息。如果没有开启开发者模式和安装驱动程序，在 CMD 窗口是无法显示手机信息的。如图 10-12 所示。

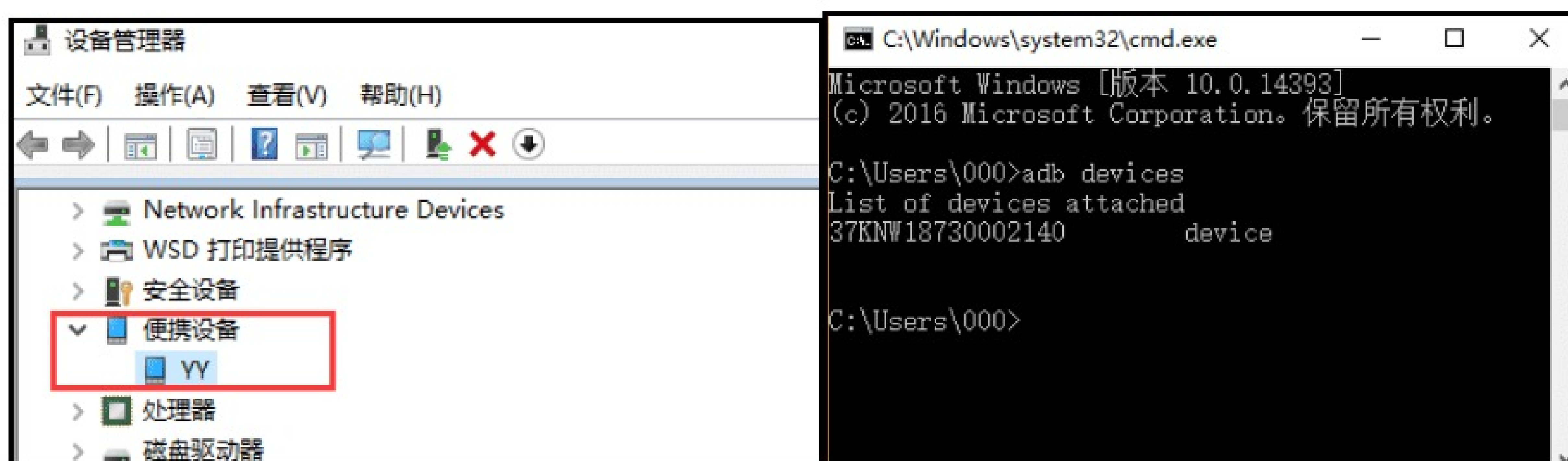


图 10-12 驱动程序（左）手机信息（右）

下一步开始搭建 Node.JS 的开发环境，它是用于运行 Appium-Server。在官方下载 Node.JS 8.12 版本，在浏览器访问 https://nodejs.org/en/download/，根据自己的计算机系统下载相应的安装包，如图 10-13 所示。

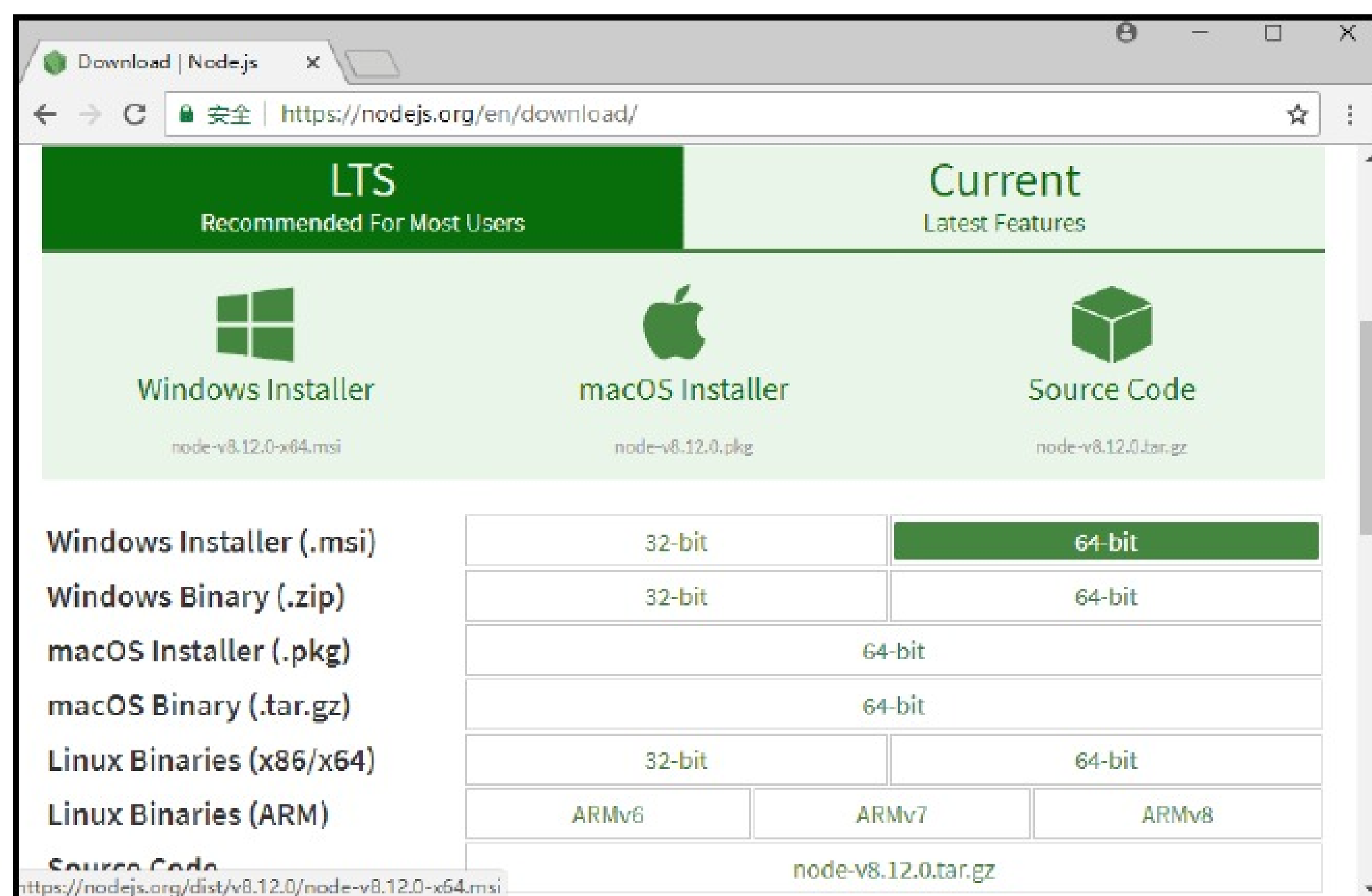


图 10-13 Node.JS 安装包

Node.JS 安装包是一个 Windows 可执行的应用程序，直接双击运行并根据安装提示进行安装，安装路径等一些安装提示使用默认设置即可。安装成功后，在 CMD 窗口验证 Node.JS 是否安装成功，验证指令以及验证结果如图 10-14 所示。



图 10-14 验证 Node.JS

Appium-Server 分为 Server 版和 Desktop 版，Server 版在 2015 年底已经停止更新，目前 Desktop 版已接替 Server 版的使命。话虽如此，Server 版现在仍然可以使用。本书以 Desktop 版为例，在 github（https://github.com/appium/appium-desktop/releases/tag/1.7.0）下载 exe 安装包，选取 1.7 最新版本，如图 10-15 所示。

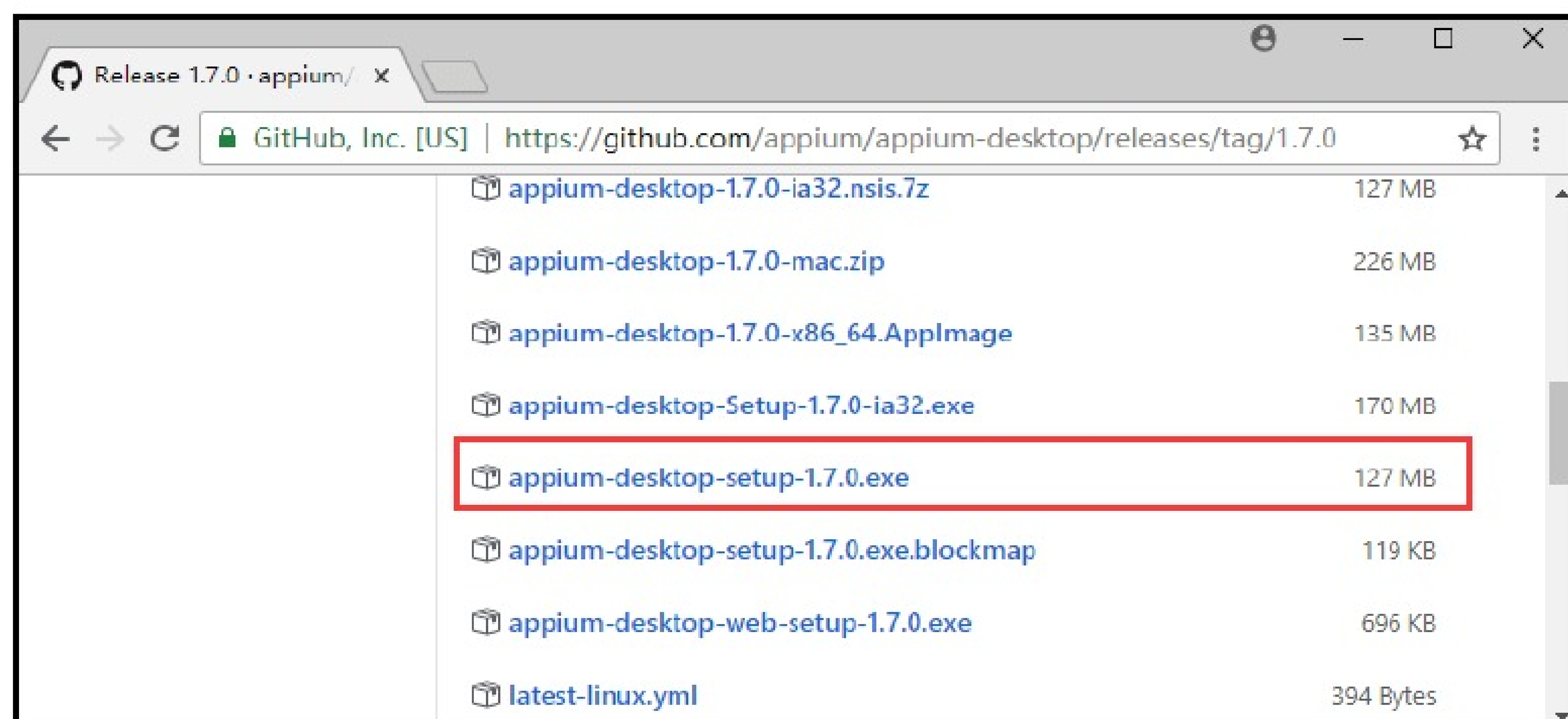


图 10-15　Appium-Server 安装包

Appium-Desktop 下载后直接运行，安装路径等一些安装提示使用默认设置即可。安装成功后在桌面上可以看到 Appium 图标，双击图标后，在 Appium-Desktop 的界面上单击“StartServerv1.9.0”按钮来启动 Appium-Server，如图 10-16 所示。

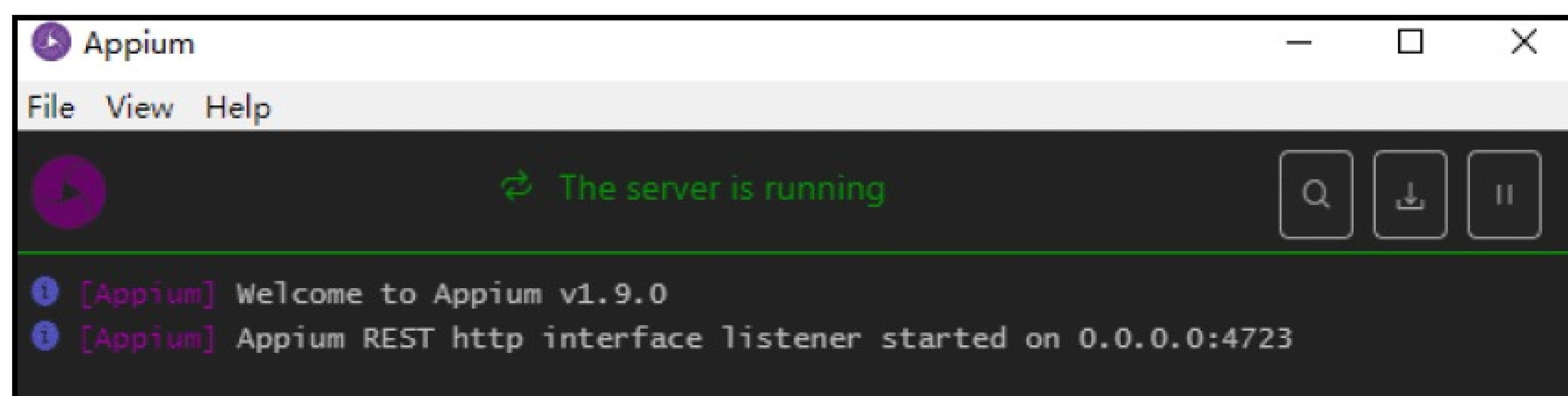


图 10-16　启动 Appium-Server

最后安装 Appium-Client 的 Python 版本，在 CMD 窗口下输入 pip install Appium-Python-Client 指令并等待安装完成即可。

在搭建 Appium 的过程中，我们分别安装了 JavaJDK、Android SDK、Node.JS、Appium-Server 和 Appium-Client，每个开发环境之间都有一定的联系，比如 JavaJDK 和 Android SDK 的兼容性问题等。

10.3　连接 Android 系统

Appium 对 Android 系统实现自动化操作，第一步是将 Appium 与 Android 进行通信连接，连接代码是相对比较固定的。在连接代码中根据 Android 系统信息进行相应的修改即可实现连接，连接代码如下：

```
from appium import webdriver
desired_caps = {}
```

```
# 设置 Android 系统信息
desired_caps['platformName'] = 'Android'
desired_caps['platformVersion'] = '8.0'
desired_caps['deviceName'] = 'huawei-lld_al20-30KNW18730002140'
desired_caps['appPackage'] = 'com.android.calculator2'
desired_caps['appActivity'] = '.Calculator'
# 向 Appium-Server 发送请求实现连接
driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_caps)
```

在上述代码中，变量 desired_caps 是一个字典，字典的 key 是代表 Appium 与 Android 系统的连接参数，字典的 value 是 Android 系统信息。每个 key 代表不同的意思，具体说明如下。

- platformName：需要被连接的操作系统，如 iOS、Android 或 FirefoxOS。
- platformVersion：Android 系统的当前版本信息，如本书的手机系统为 8.0。
- deviceName：每台移动设备或模拟器的设备名，设备名是唯一的。
- appPackage：需要执行自动化的 Android 应用的包名。
- appActivity：Android 应用包中启动的 Android Activity 名称。

这 5 个参数是连接 Android 系统的基本参数，每个参数值的获取方式各不相同。下面我们讲述参数值的获取方法。

参数 platformName 只有三个参数值，分别是 iOS、Android 和 FirefoxOS，代表不同的操作系统。

参数 platformVersion 是移动设备或模拟器的系统版本信息。以华为手机的系统版本信息获取为例，在手机的“设置”→“系统”→“关于手机”里面找到 Android 版本信息，如图 10-17 所示。

图 10-17　Android 版本信息

参数 deviceName 的参数值获取较为繁琐，获取过程需要借助工具来完成。打开 Android SDK 所在的文件夹，找到 tools 文件夹里的 uiautomatorviewer.bat 文件并双击运行，该文件启动一个名为 UI Automator Viewer 的软件，该软件用于捕捉 Android 应用程序的控件元素信息，在下一节中还需要借助该软件来实现元素的定位。软件界面如图 10-18 所示。

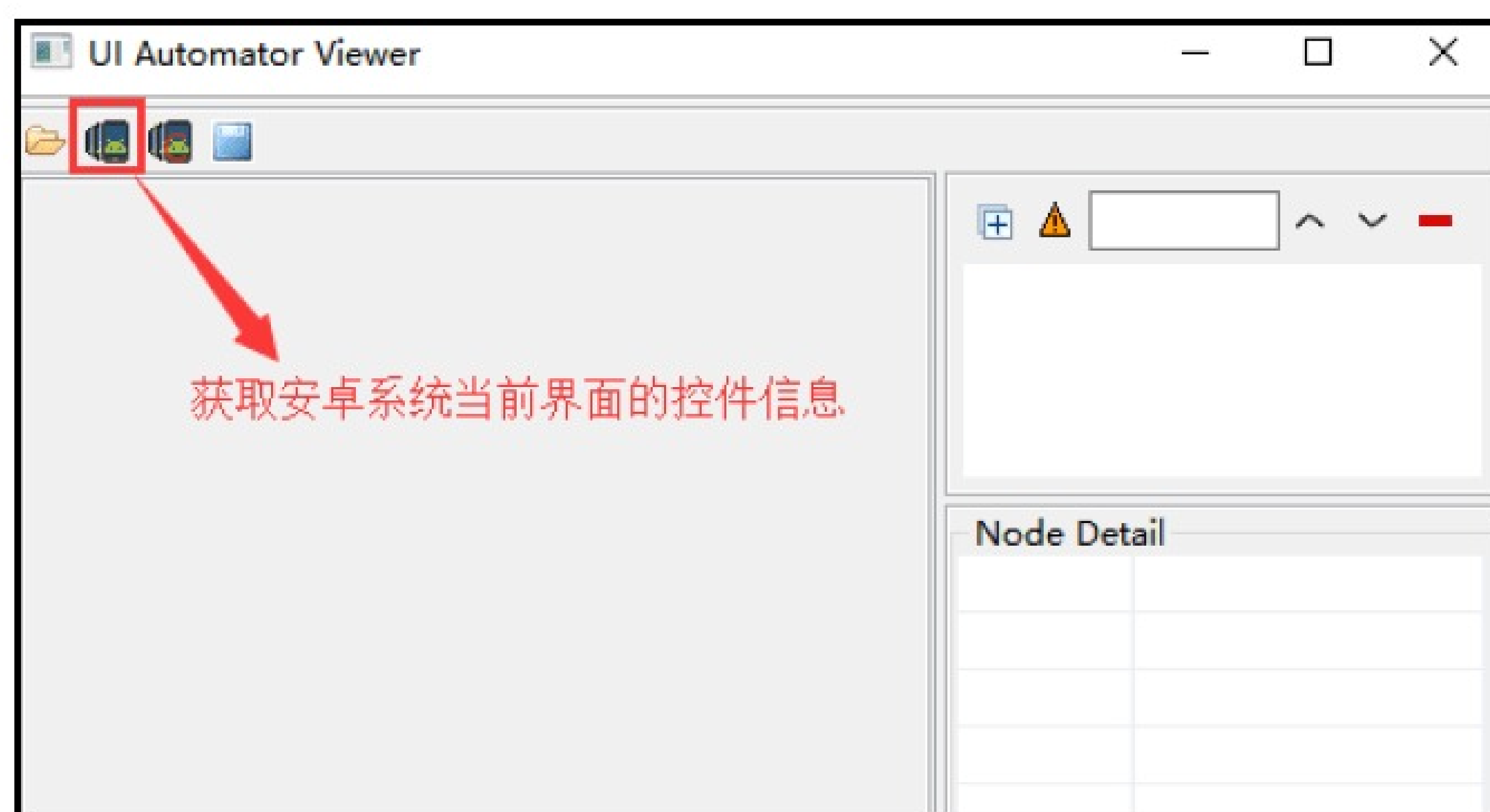


图 10-18　软件界面

在 Android SDK 的文件路径找到 AVD Manager.exe 并双击运行，该 exe 程序可以启动 Android 模拟器，具体的启动方式如图 10-11 所示。再将手机连接到计算机，连接之前确保手机已开启 USB 调试模式，连接成功后，手机界面会出现一个 USB 调试提示信息，单击“确定”按钮即可，如图 10-19 所示。



图 10-19　USB 调试提示信息

现在计算机已分别开启了 Android 模拟器和连接了一台 Android 手机，单击图 10-18 所标注的按钮，软件就会出现一个设备选择的界面，界面中的设备名就是参数 deviceName 的参数值。总的来说，参数 deviceName 的获取必须借助工具 UI Automator Viewer，同时保证计算机已连接两台或以上的 Android 设备或 Android 模拟器，如图 10-20 所示。

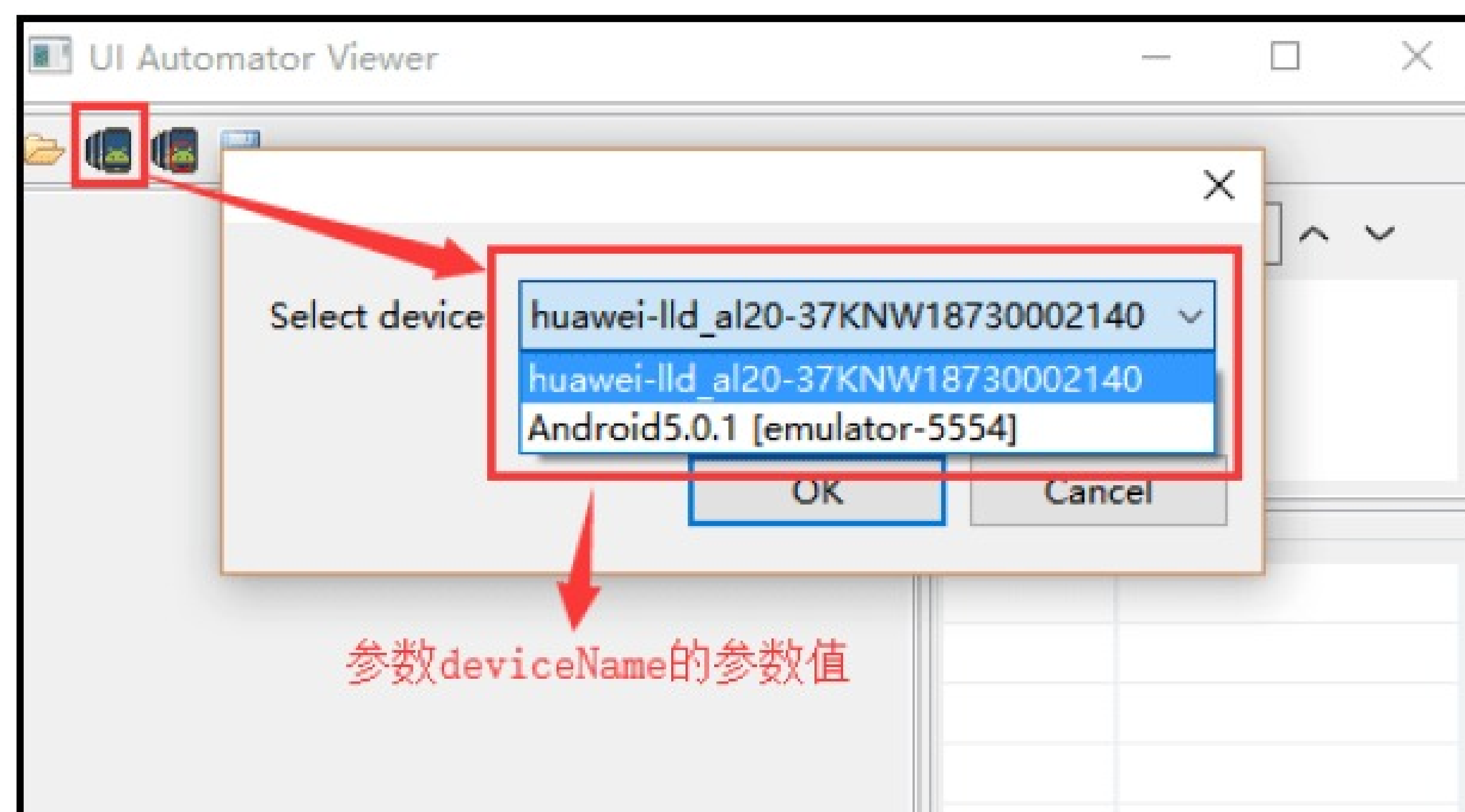


图 10-20　获取 deviceName

参数 appPackage 同样需要借助工具 UI Automator Viewer 获取，选中图 10-20 的 huawei 设备并且确保手机的屏幕常亮，单击“OK”按钮，软件就会自动捕捉手机当前界面的控件信息。单击手

机上的某个控件，该控件信息就会显示在右侧。其中参数 package 的参数值就是参数 appPackage 的参数值，如图 10-21 所示。

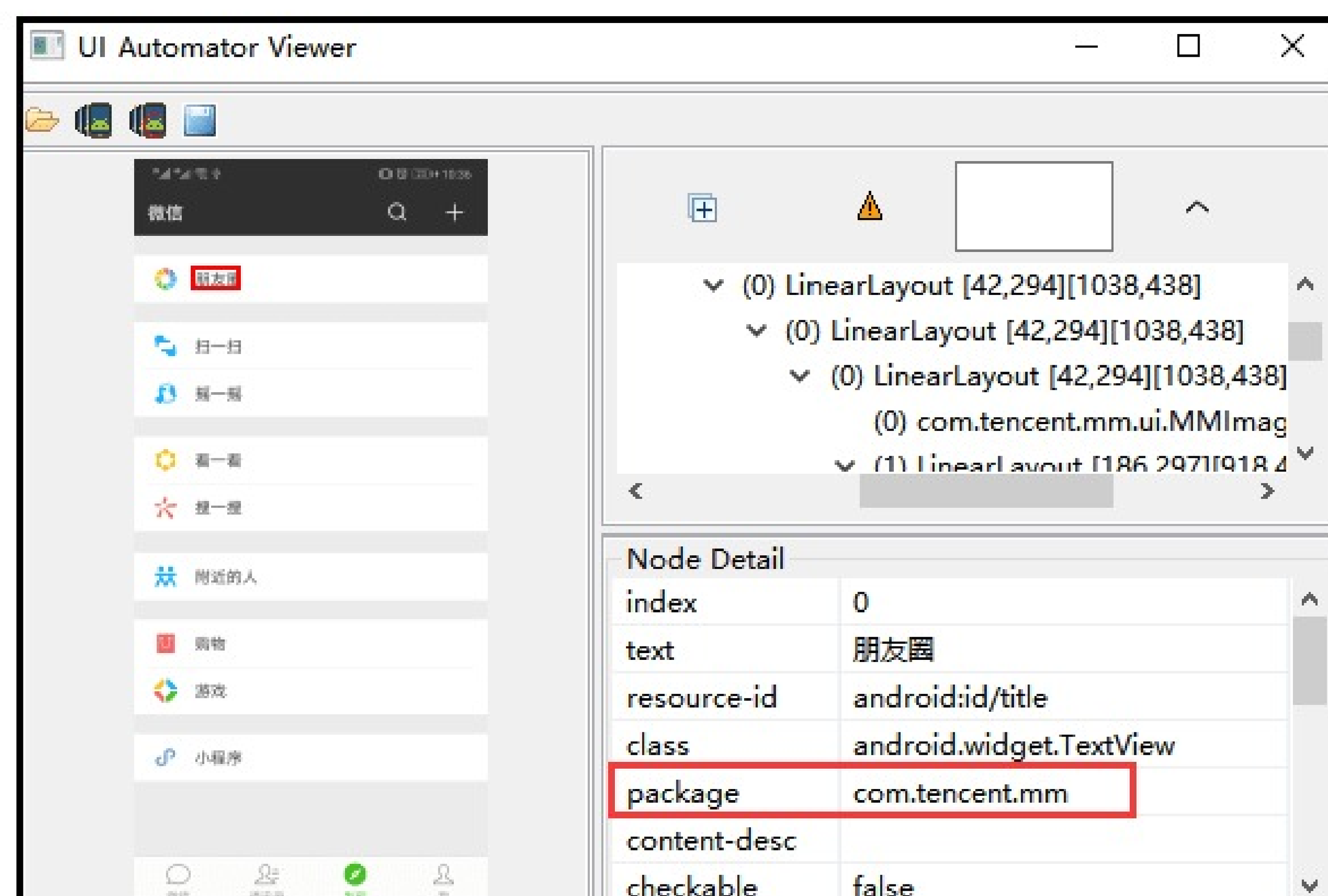


图 10-21　控件信息

参数 appActivity 的获取需要保证计算机上只有一台 Android 设备或 Android 模拟器。以手机为例，关闭 Android 虚拟机，打开 CMD 窗口并输入 adb shell dumpsys activity activities 指令来获取当前设备的程序运行信息。在这些信息中可以找出 appActivity 的参数值，比如查找微信的 appActivity，通过参数 appPackage 确定 appActivity 的参数值，如 realActivity=com.tencent.mm/.ui.LauncherUI，斜杠后面的内容.ui.LauncherUI 就是参数 appActivity 的参数值，如图 10-22 所示。

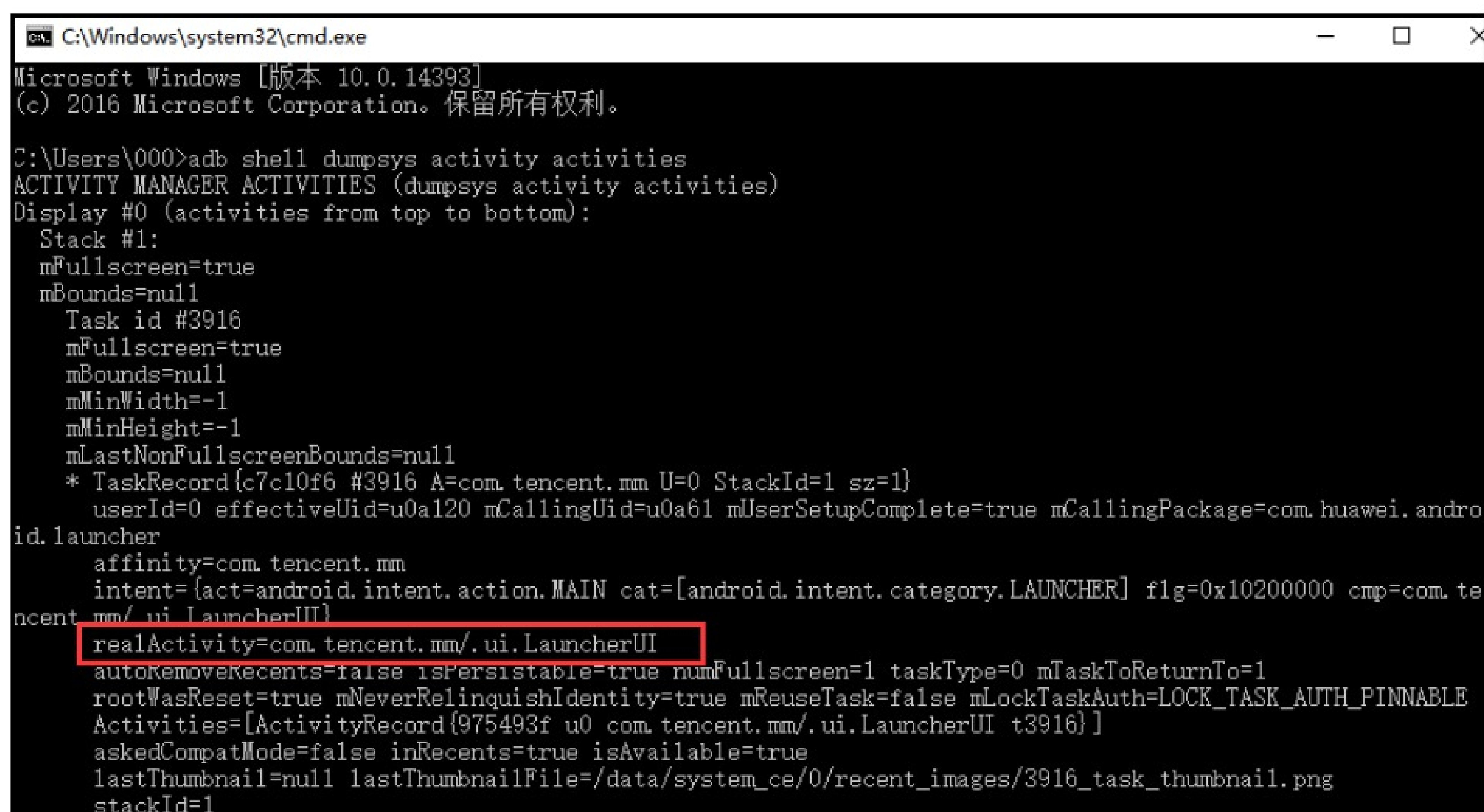


图 10-22　查找 appActivity

参数 appPackage 和参数 appActivity 的获取方法并不是唯一的，这两个参数都可以通过不同的方法获取，有兴趣的读者可以自行在网上查阅相关的资料。除此之外，Appium 在连接移动设备或模拟器上还提供了很多连接参数，本书列出一些常用的参数及其说明，如表 10-1 所示。

表 10-1 常用参数及说明

参数	说明	参数值
	通用参数	
automationName	选择自动化引擎	Appium（默认）、Selendroid、UiAutomator2、Espresso
app	在移动设备上安装应用程序	安装包存放路径，如 D:\QQ.apk
browserName	移动网页浏览器的名称	如 iOS 的“Safari”，Android 的“Chrome”
newCommandTimeout	客户端退出并结束连接之前，Appium 等待客户端新命令的时间	时间以秒为单位
language	设置语言	如中国——CN
locale	设置语言环境	如中文——zh_CN
udid	连接物理设备的唯一设备标识符	如手机的序列号
noReset	连接之前不重置应用程序状态	布尔型，默认 True
fullReset	执行完整的重置，即清除应用数据并卸载 apk	布尔型，默认 False
eventTimings	启用或禁用各种 Appium 内部事件的时间报告	默认为 False
enablePerformanceLogging	启用 Chromedriver（在 Android 上）或 Safari（在 iOS 上）性能记录	默认 False
	仅限 Android	
appWaitActivity	等待 Android 应用程序启动	等同 appActivity 参数值
appWaitPackage	等待 Android 应用程序的程序包	等同 appPackage 参数值
appWaitDuration	设置 appWaitActivity 启动的超时	以毫秒为单位，默认值为 20000
deviceReadyTimeout	设置设备准备状态的超时	以秒为单位
androidInstallTimeout	等待 apk 安装到设备的超时	以毫秒为单位，默认为 90000
androidInstallPath	设置 apk 的安装路径	默认路径：/data/local/tmp
adbPort	用于连接到 ADB 服务器的端口	默认 5037
chromeOptions	允许 ChromeDriver 传递 chromeOptions 功能	chromeOptions: {args: ['--disable- popup-blocking']}
recreateChromeDriverSessions	在移至非 ChromeDriver 网页浏览的情况下杀死 ChromeDriver 会话	默认为 False
networkSpeed	设置网络速度模拟。指定最大的网络上传和下载速度	默认为 full
androidScreenshotPath	设置设备上的屏幕截图的路径地址	默认路径：/data/local/tmp
resetKeyboard	使用 unicode 编码方式发送字符串	布尔型，默认值 False
unicodeKeyboard	将键盘隐藏起来	布尔型，默认值 False
	仅限 iOS	
calendarFormat	设置日历格式	例如 gregorian
udid	连接物理设备的唯一设备标识符	如手机的序列号

（续表）

参数	说明	参数值
	仅限 iOS	
locationServicesEnabled	强制定位服务处于打开或关闭状态	布尔型，默认保持当前的模拟设置
locationServicesAuthorized	将位置服务设置为授权或未授权	布尔型，默认保持当前的模拟设置
safariInitialUrl	初始 Safari 浏览器网址	默认为本地欢迎页面
safariAllowPopups	允许 JS 在 Safari 中打开新窗口	布尔型，默认保持当前的模拟设置
safariIgnoreFraudWarning	防止 Safari 显示欺诈网站警告	布尔型，默认保持当前的模拟设置
safariOpenLinksInBackground	Safari 是否允许在新窗口中打开链接	布尔型，默认保持当前的模拟设置
appName	应用程序的显示名称	例如 UICatalog

10.4　App 的元素定位

上一章节讲述了 Appium 连接 Android 系统的实现过程，程序中以 driver 对象表示连接成功并且将连接状态持久化，整个自动化程序都是围绕这个 driver 对象进行展开。Appium 为 driver 对象提供了许多函数方法，每个函数方法是实现某个自动化操作。

由于 Appium 是在 Selenium 的基础上进行封装，所以 Appium 的元素定位与操作采用了 Selenium 部分的方法。在讲述元素定位与操作之前，我们先学习元素的查找方法，Android 系统的元素查找需要借助软件 UI Automator Viewer 实现。以手机的计算器为例，比如查找数字 6 的元素属性，具体操作步骤如下：

步骤01 将手机与计算机进行连接，连接之前确保手机已开启 USB 调试模式。

步骤02 唤醒手机屏幕，当手机界面出现 USB 调试提示信息时，单击“确定”按钮并打开手机的计算器。

步骤03 打开软件 UI Automator Viewer，单击“DeviceScreenshot”按钮捕捉手机当前界面。

步骤04 捕捉成功后，在软件的左侧会出现手机界面的截图，单击截图的数字 6，该数字的相关属性都会展示在软件的右侧，这些属性就是我们所需的元素属性，如图 10-23 所示。

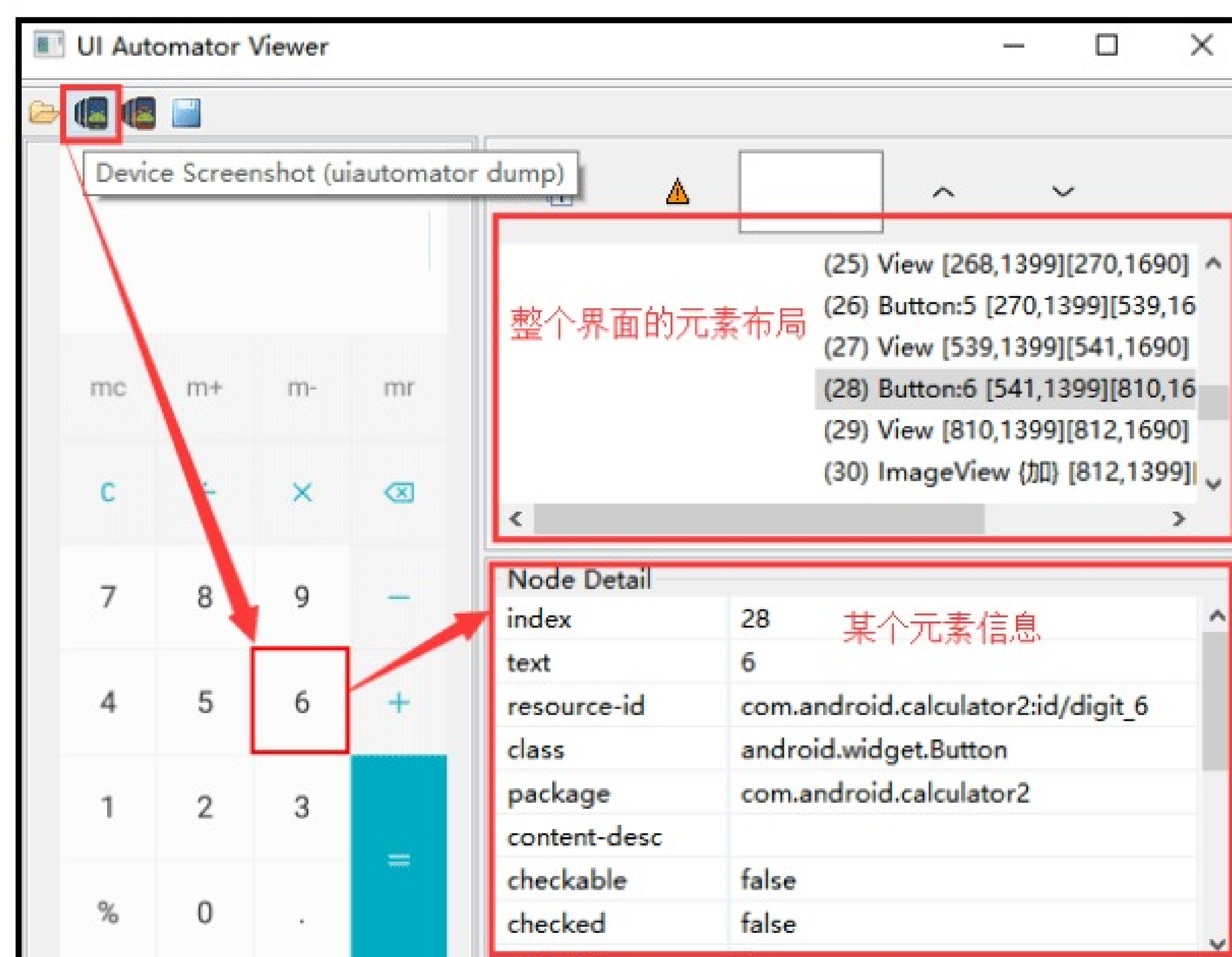


图 10-23　元素查找

数字 6 的元素属性一共有 17 个，但是只有 5 个属性能用于元素定位，它们分别是 index、text、resource-id、class 及 content-desc。那么，Appium 对数字 6 的定位方法如下：

```
#通过 index 定位
#Appium 的 uiautomator 方法
index = '28'
ua = 'new UiSelector().index(' + index + ')'
driver.find element by android uiautomator(ua).click()

#通过 text 定位
#Appium 的 uiautomator 方法
text = '6'
ua = 'new UiSelector().text("' + text + '")'
driver.find element by android uiautomator(ua).click()

#通过 resource-id 定位
resourceId = 'com.android.calculator2:id/digit 6'
#Selenium 的方法
driver.find element by id(resourceId)
#Appium 的 uiautomator 方法
ua = 'new UiSelector().resourceId("' + resourceId + '")'
driver.find element by android uiautomator(ua).click()

#通过 class 定位
#Selenium 的方法
class name = 'android.widget.Button'
driver.find element by class name(class name)

#通过 content-desc 定位
#Appium 的 uiautomator 方法
#由于数字 6 的属性值为空，此处选取按键 C
description = '清除'
ua = 'new UiSelector().description("' + description + '")'
driver.find element by android uiautomator(ua).click()
#方法二
```

```
    driver.find_element_by_accessibility_id('清除').click()

    # Xpath 定位
    xpath = '//android.widget.Button[contains(@text,"6")]'
    driver.find_element_by_xpath(xpath).click()
```

元素定位主要使用了 Selenium 的方法和 Appium 的 uiautomator 方法实现，在 5 个属性中，除了元素属性 class 之外，其余 4 个元素属性都能使用 Appium 的 uiautomator 方法进行定位，Selenium 的方法只适用于 class 和 resource-id 属性，而 Selenium 的 Xpath 方法是根据元素的布局进行定位，它能用于任何 Android 应用程序。在 PyCharm 编写代码的时候，代码提示还会出现所有 Selenium 的定位方法，这些定位方法主要用于手机浏览器的网页自动化开发。

使用 Appium 的 uiautomator 方法进行元素定位的时候，不同的属性有不同的代码编写规则，具体的差异体现在上述代码的变量 ua 上，由于变量 ua 的代码格式较为固定，只要细心观察才能发现差异之处。对于 Xpath 定位，需要掌握 Xpath 语法才能写出相应的定位代码，由于本书篇幅有限，此处就不做详细介绍，有兴趣的读者可以自行查阅资料。

10.5 App 的元素操控

在讲述元素定位的时候，定位后的元素都执行了单击处理，该操作由 click()方法实现。当我们使用手机的时候，使用过程中大多数操作都是单击、文本输入和滑动。单击是由 click()方法实现的；文本输入由 send_keys()方法实现；滑动操作由 swipe()方法实现。单击操作在上一节的代码中已有使用示例，并且使用方法相对简单，此处不再赘述，下面主要讲述文本输入和滑动操作。

以美团为例，单击首页顶部的搜索文本框会进入一个搜索页面，然后可在搜索页面中输入相关的搜索内容，如图 10-24 所示。

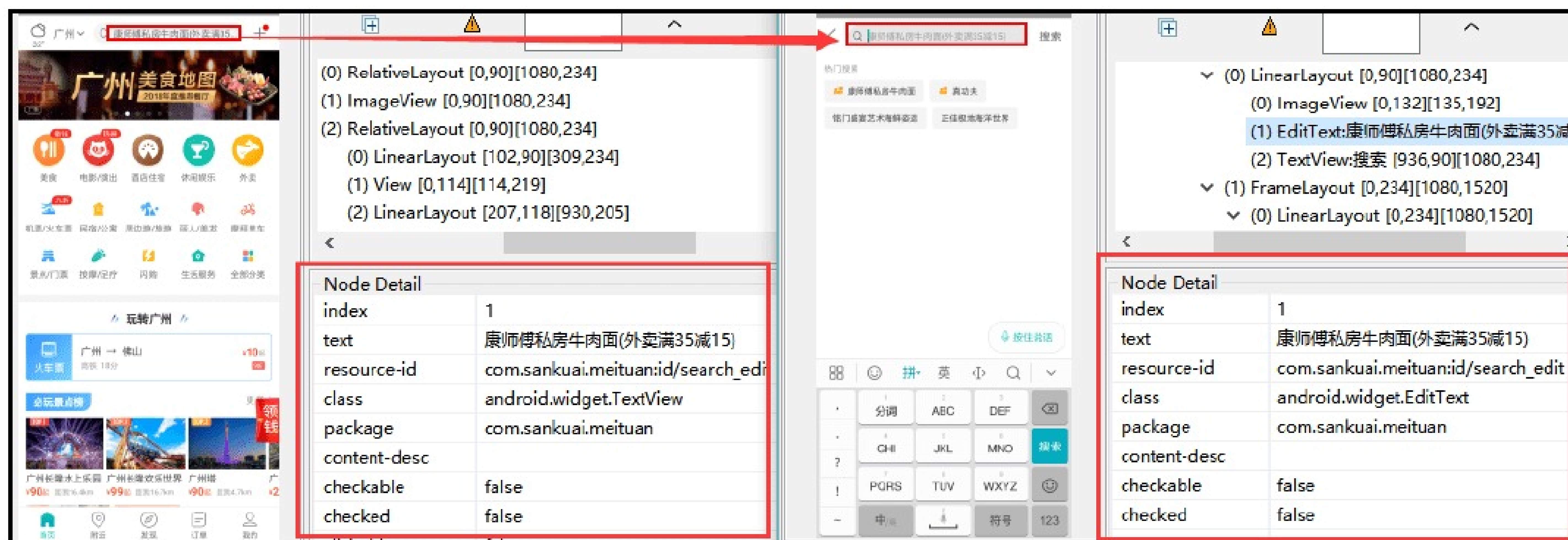


图 10-24 查找元素信息

我们要对图中两个文本框进行定位并操控，第一个文本框是进行单击操控，第二个文本框是进行文本输入操作，具体的实现代码如下：

```
    from appium import webdriver
    import time
    desired_caps = {
```

```
        'platformName': 'Android',
        'platformVersion': '8.0',
        'deviceName': 'huawei-lld al20-30KNW18730002140',
        'appPackage': 'com.sankuai.meituan',
        'appActivity': 'com.meituan.android.pt.homepage.activity.MainActivity',
        # 设置中文输入
        'unicodeKeyboard': True,
        'resetKeyboard': True,
    }
    # 向 Appium-Server 发送请求实现连接
    driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_caps)
    time.sleep(3)
    # 单击系统提示框
    for i in range(2):
        resourceId = 'com.android.packageinstaller:id/permission_allow_button'
        driver.find_element_by_id(resourceId).click()
        time.sleep(3)
    # 单击首页输入框
    resourceId = 'com.sankuai.meituan:id/search_edit'
    driver.find_element_by_id(resourceId).click()
    time.sleep(3)
    # 输入搜索内容
    resourceId = 'com.sankuai.meituan:id/search_edit'
    driver.find_element_by_id(resourceId).send_keys('广州长隆')
```

在代码中，字典 desired_caps 额外设置了参数 unicodeKeyboard 和 resetKeyboard，前者是将键盘输入内容改为 unicode 格式，后者是将手机的输入法改为 Appium 的输入法。只有同时设置这两个参数，Appium 才能在手机上输入中文内容，否则输入的内容就会变成乱码。

Appium 在运行 Android 应用程序的时候，应用程序在启动时是处于一种初始化的状态，也就是说 Appium 清除了用户在这个应用上的使用痕迹。当 Android 应用程序启动成功后，系统会出现相应的系统提示框，因此在执行自动化操作之前，还需要对这些系统提示进行相应的处理才能执行下一步的操作，如图 10-25 所示。



图 10-25　系统提示框

Appium 的滑动操作可以分为上滑、下滑、左滑和右滑，不管哪一种滑动，它们都是由 swipe()方法实现，只要对 swipe()方法传入不同的参数就能实现不同的滑动方式，swipe()方法的定义如下：

```
    swipe (int start_x, int start_y, int end_x, int y, duration)
    参数说明:
    int start_x 开始滑动的 x 坐标
    int start_y  开始滑动的 y 坐标
```

```
    int end x   结束点 x 坐标
    int end y   结束点 y 坐标
    duration 滑动时间（默认 5 毫秒）
```

从 swipe()方法定义可以看到，滑动屏幕需要借助屏幕上的坐标位置，由于每台手机的分辨率和尺寸大小不同，如果将滑动位置设为一个固定的坐标，在其他手机上不一定能适用，所以只能够根据手机的屏幕大小来制定滑动位置。Appium 提供了相应的方法来获取手机屏幕的尺寸大小，实现过程如下：

```
# 获得手机屏幕分辨率 x,y
def getSize():
    x = driver.get_window_size()['width']
    y = driver.get_window_size()['height']
    return (x, y)
```

函数 getSize()是我们自定义的函数，在函数中使用了 Appium 的 get_window_size()方法来获取手机屏幕分辨率。每台手机的坐标点都是从左上方为起点，右下方为终点，这与计算机屏幕分辨率的坐标点分布原理是相同的，如图 10-26 所示。

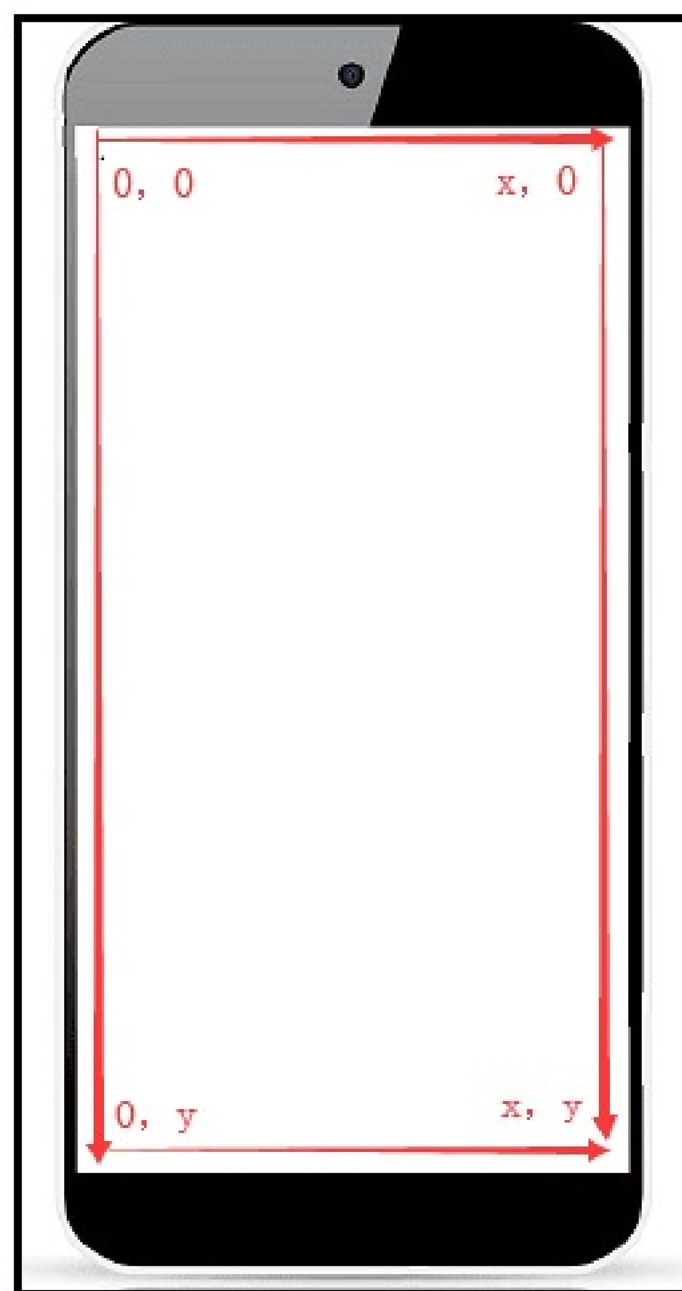


图 10-26　手机屏幕分辨率

滑动屏幕主要是在屏幕的正中位置进行的，从函数 getSize()的返回值可以计算不同位置的坐标点，有了这些坐标点就可以实现屏幕滑动，每种滑动方式的实现代码如下：

```
# 向上滑动
def swipeUp(t):
    local = getSize()
    x = int(local[0] * 0.5)
    y1 = int(local[1] * 0.75)
    y2 = int(local[1] * 0.25)
    driver.swipe(x, y1, x, y2, t)

# 向下滑动
def swipeDown(t):
    local = getSize()
    x = int(local[0] * 0.5)
```

```
        y1 = int(local[1] * 0.25)
        y2 = int(local[1] * 0.75)
        driver.swipe(x, y1, x, y2, t)

    # 向左滑动
    def swipLeft(t):
        local = getSize()
        x1 = int(local[0] * 0.75)
        y = int(local[1] * 0.5)
        x2 = int(local[0] * 0.05)
        driver.swipe(x1, y, x2, y, t)

    # 向右滑动
    def swipRight(t):
        local = getSize()
        x1 = int(local[0] * 0.05)
        y = int(local[1] * 0.5)
        x2 = int(local[0] * 0.75)
        driver.swipe(x1, y, x2, y, t)
```

不同的滑动方式对 swipe()的参数有不同的设置。比如向上滑动，X 坐标的起始位置与结束位置是固定不变的，Y 坐标的起始位置是屏幕的 3/4 位置，结束位置是屏幕的 1/4 位置，也就是从下往上滑动，如图 10-27 所示。每个函数的参数 t 代表滑动时间，参数值的大小会直接影响滑动效果，一般设置为 1000，如果使用 swipe()的默认值 5 毫秒，则在手机上完全没有滑动效果。



图 10-27　手机屏幕位置

除了上述的自动化操作之外，Appium 还提供了许多实用的操作方法。这些方法都是由 driver 对象使用，它们定义在 Python 安装目录\Lib\site-packages\appium\webdriver\webdriver.py 中，每种方法所实现的功能以及参数都有注释说明，有兴趣的读者可以自行查阅。

10.6　实战：淘宝商品采集

通过前面的学习，相信大家对 Appium 的自动化功能有了一定的了解和掌握，在本节中，我们以手机淘宝的商品信息采集为例，进一步掌握 Appium 的开发。整个项目的业务流程大致如下：

（1）Appium 启动手机淘宝 App，并处理 Android 系统的提示信息。

（2）单击淘宝首页顶部的搜索框，进入淘宝的搜索界面。

（3）在搜索界面输入搜索内容并单击“搜索”按钮。

（4）进入商品界面，单击“销量”按钮，将商品以销量排序。

（5）读取当前界面的商品信息，对每条信息进行去重和写入处理。

（6）在商品界面执行向上滑动，读取其他商品信息，重复执行步骤（5）。

分析上述的业务流程，在手机淘宝 App 里需要定位的元素分别有：淘宝首页搜索框、搜索界面的搜索框、商品信息界面的“销量”按钮以及商品信息的标题和价格。在软件 UI Automator Viewer 里分别定位并查找这些元素信息，若软件在截取手机界面时出现报错，请先关闭 Appium 服务器再次执行截取操作，因为 Appium 服务器会对软件的使用有一定的影响。

打开手机淘宝，使用软件 UI Automator Viewer 截取整个淘宝首页的元素信息，单击软件截图的搜索框，发现搜索框可以通过 index、resource-id 和 class 属性进行定位，如图 10-28 所示。元素的 text 属性虽然有属性值，但是每次打开淘宝都会发现 text 的属性值各不相同。属性 class 可以在一个界面里重复使用，但是此界面只有一个搜索框，因此 class 属性也能实现定位。在选择属性进行定位的时候，需要结合实际情况来分析每个属性是否可行。以 resource-id 定位为例，代码如下所示：

```
# 单击首页搜索框
resourceId = 'com.taobao.taobao:id/home_searchedit'
driver.find_element_by_id(resourceId).click()
```

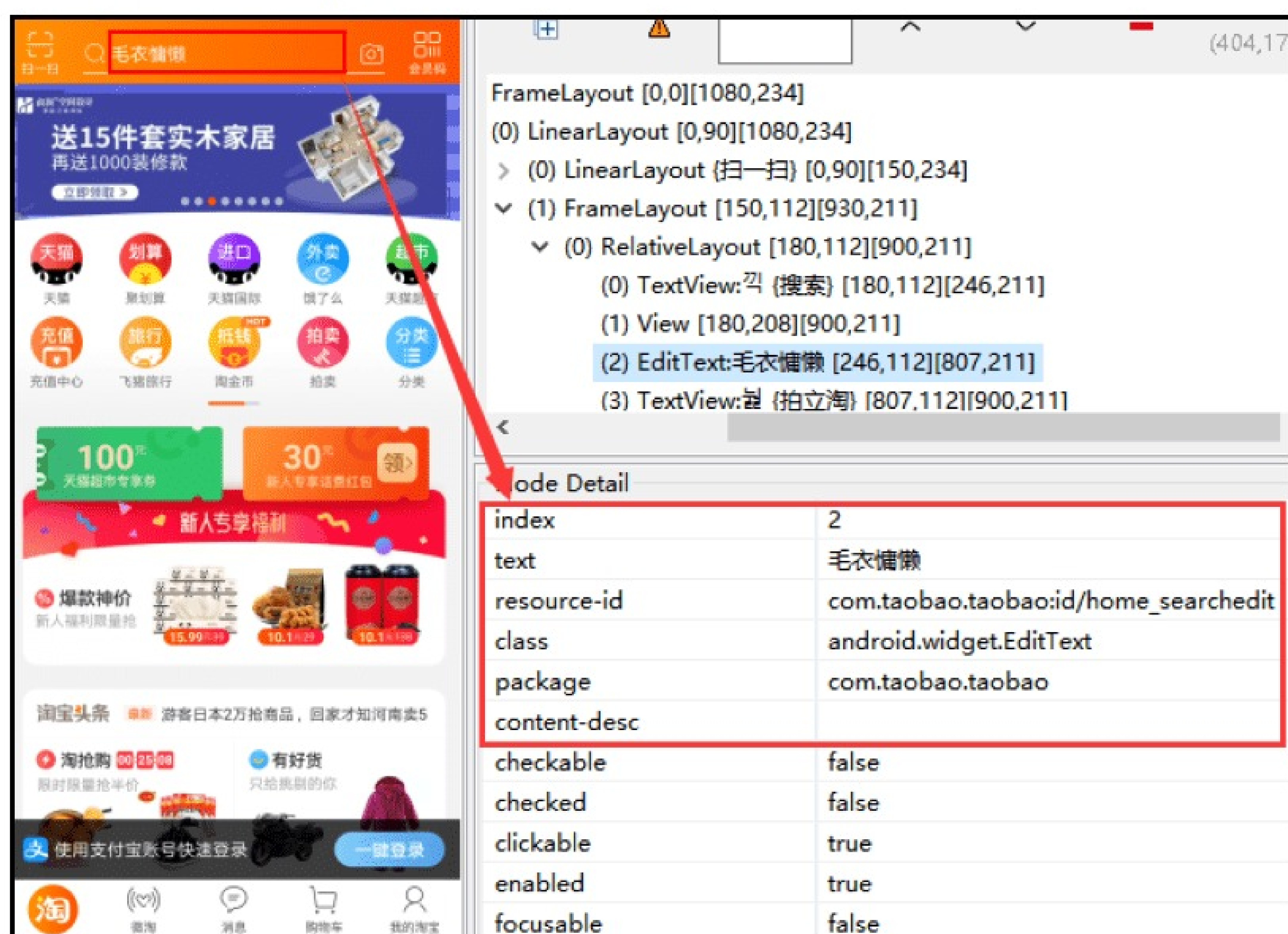


图 10-28　淘宝首页搜索框

单击首页的搜索框后就会进入到搜索界面，搜索界面的搜索框与首页的搜索框是不同的元素，需要重新对搜索界面的搜索框进行信息截取，如图 10-29 所示。在搜索框中输入商品的关键词，并单击搜索框右侧的“搜索”按钮就能搜索相关的商品信息。搜索框和“搜索”按钮的定位以 resource-id 为例，实现代码如下：

```
# 输入搜索内容
text = '玩转 Python 网络爬虫'
resourceId = 'com.taobao.taobao:id/searchEdit'
driver.find_element_by_id(resourceId).send_keys(text)
# 点击搜索按钮
resourceId = 'com.taobao.taobao:id/searchbtn'
driver.find_element_by_id(resourceId).click()
```

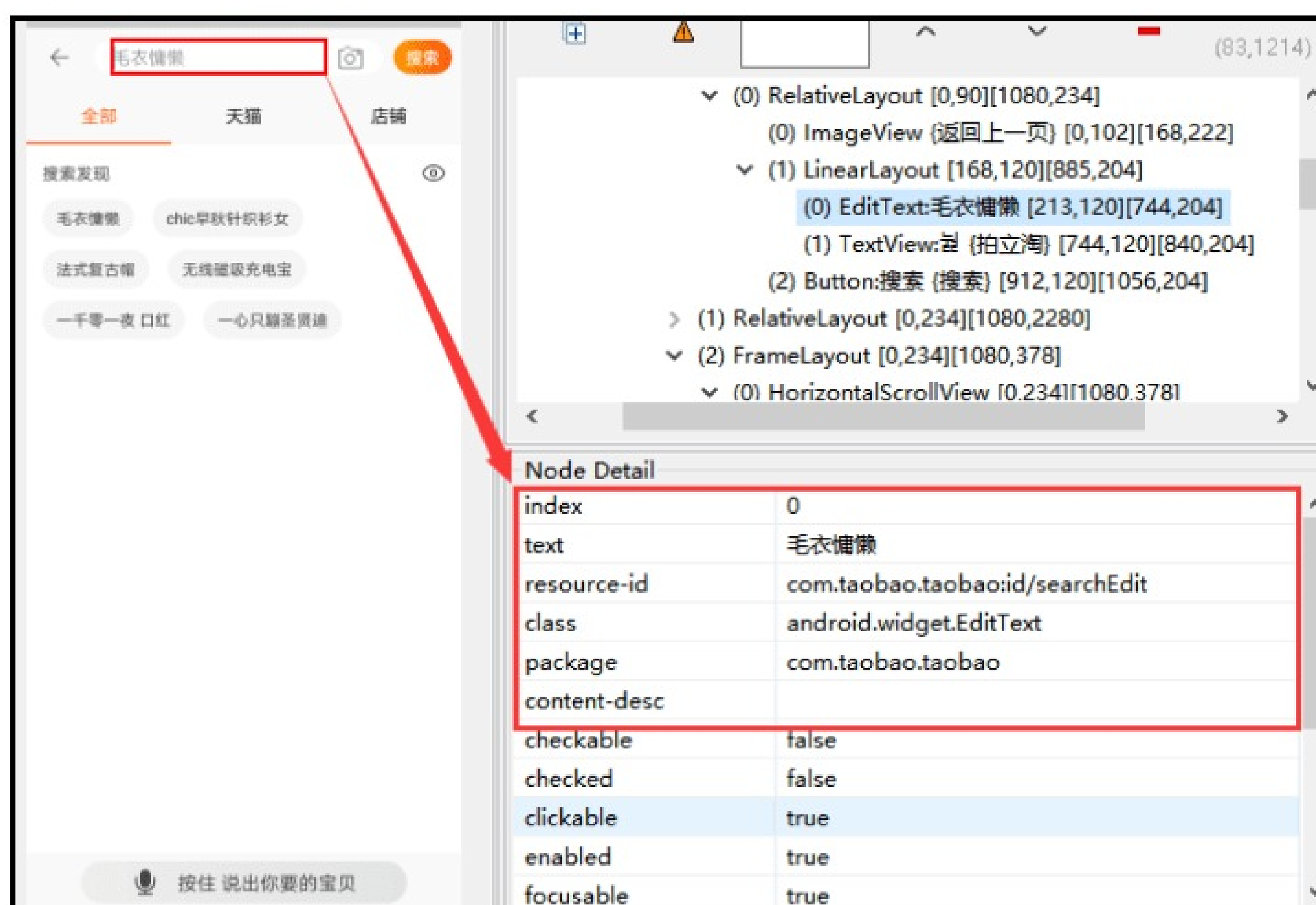


图 10-29　搜索界面的搜索框

搜索所得的商品信息显示在商品界面，在该界面上单击“销量”按钮，将所有的商品按照销量的大小重新排序，从图 10-30 得知，“销量”按钮的属性 text 相比其他属性较为稳定而且具有唯一性，因此该按钮以属性 text 进行定位，代码如下所示：

```
# 单击销量排序
sales = 'new UiSelector().description("销量")'
driver.find_element_by_android_uiautomator(sales).click()
```

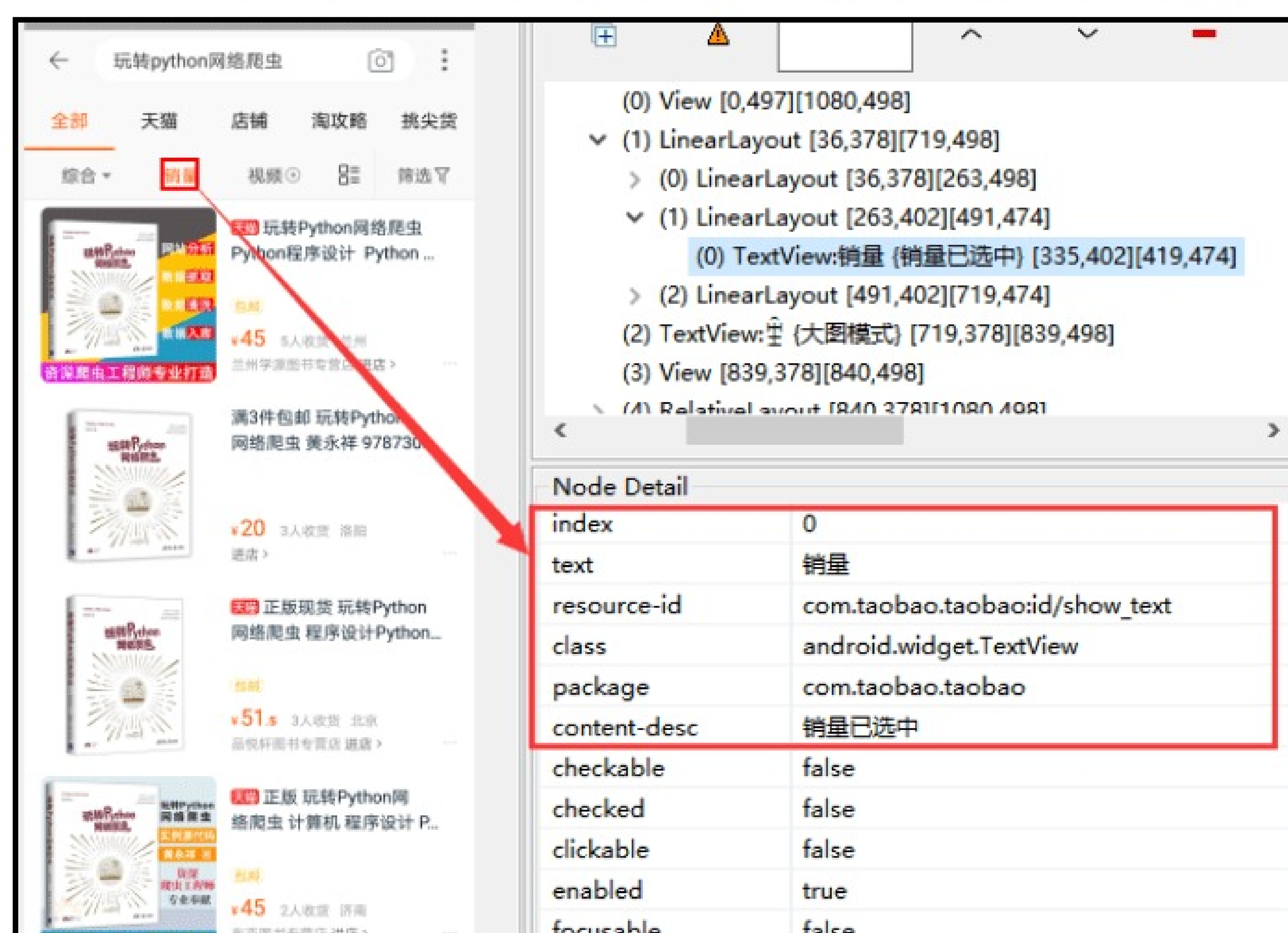


图 10-30　商品界面的“销量”按钮

我们对排序后的商品进行信息采集，将每条商品的标题和价钱写入到一个新的字典里，再将这个字典存放到一个列表中。从图 10-31 看到，每条商品以 RelativeLayout 元素为单位，每个 RelativeLayout 元素都包含了商品的标题、价钱以及运费等信息。因此先定位所有的 RelativeLayout 元素，再对这些元素进行遍历处理，每次遍历获取相应的标题和价钱，具体的代码如下：

```
MyList = []
#滑动屏幕 5 次
for t in range(5):
#定位所有 RelativeLayout 元素
    resourceId = 'com.taobao.taobao:id/auction_layout'
    info = driver.find elements by id(resourceId)
    check and delay()
#遍历每个 RelativeLayout 元素
    for i in info:
        try:
            MyDict = {}
            # 获取标题
            resourceId = 'com.taobao.taobao:id/title'
            title = i.find element by id(resourceId)
            MyDict['title'] = title.text.strip()
            # 获取价格
            resourceId = 'com.taobao.taobao:id/priceBlock'
            price = i.find_element_by_id(resourceId)
            MyDict['price'] = price.get attribute("contentDescription")
            # 去重并写入列表
            if MyDict not in MyList:
                MyList.append(MyDict)
        except: pass
    # 滑动屏幕
    swipeUp(1000)
```

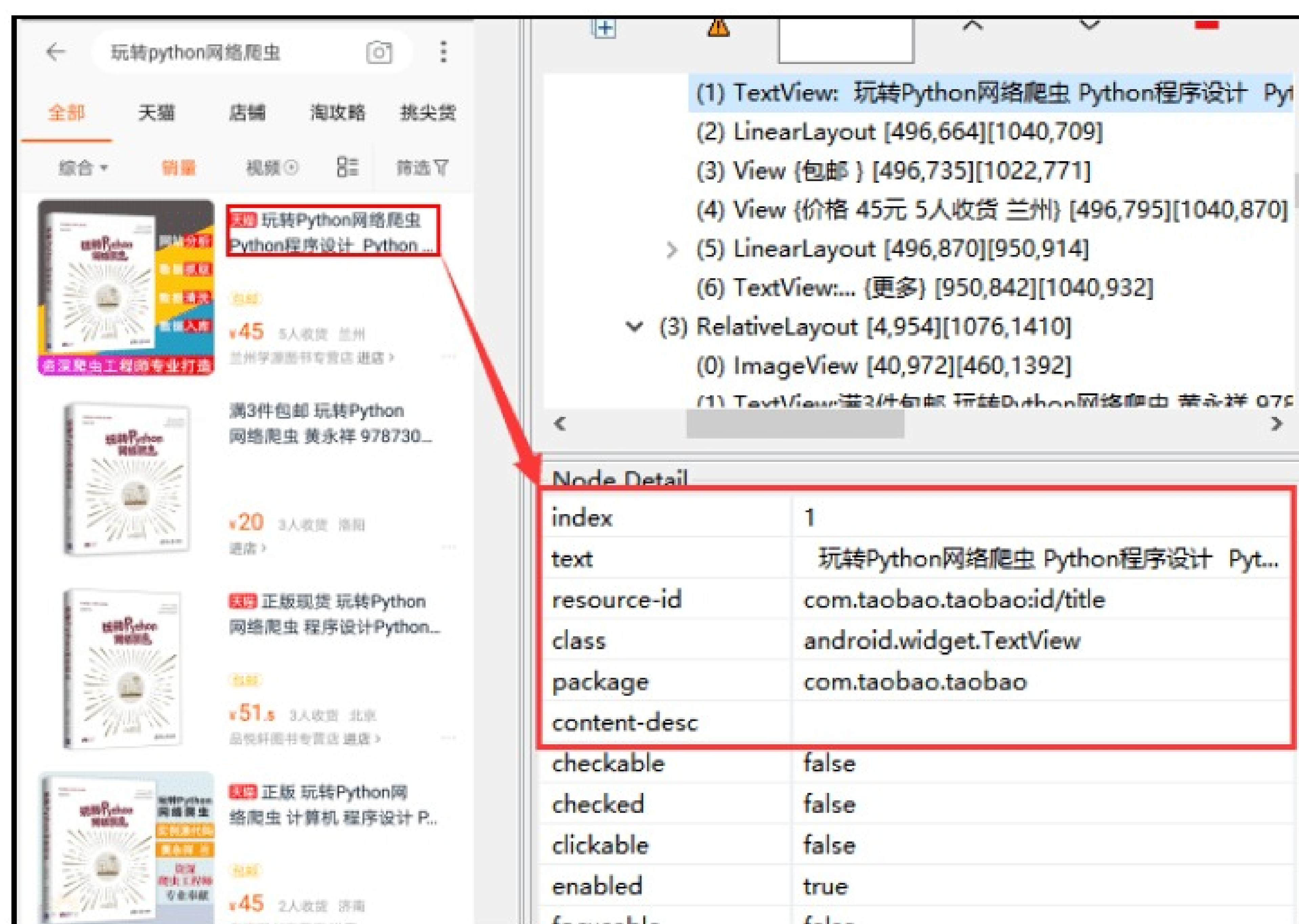


图 10-31　商品信息的标题

在每个元素之间加入延时等待，因为商品的搜索和销量排序都是从淘宝的服务器获取数据，这个获取过程会涉及网络延时，所以加入延时功能是为了更好地协调自动化操作与应用程序的响

应，使两者尽量保持同步执行。

除此之外，Appium 在启动 Android 应用的时候，在应用界面中都会出现系统提示框，我们将提示框的处理和延时功能都定义在一个函数里实现。综合上述分析，整个项目的功能代码如下：

```
from appium import webdriver
import time
# 延时与检测系统提示
def check_and_delay(ts=10):
    time.sleep(ts)
    try:
        driver.find_element_by_id('android:id/button1').click()
    except: pass

# 获得屏幕坐标 x,y
def getSize():
    x = driver.get_window_size()['width']
    y = driver.get_window_size()['height']
    return (x, y)

# 屏幕向上滑动
def swipeUp(t):
    local = getSize()
    x = int(local[0] * 0.75)
    y1 = int(local[1] * 0.75)
    y2 = int(local[1] * 0.25)
    driver.swipe(x, y1, x, y2, t)

if __name__ == '__main__':
    desired_caps = {
        'platformName': 'Android',
        'platformVersion': '8.0',
        'deviceName': 'huawei-lld_al20-30KNW18730002140',
        'appPackage': 'com.taobao.taobao',
        'appActivity': 'com.taobao.tao.homepage.MainActivity3',
        # 设置中文输入
        'unicodeKeyboard': True,
        'resetKeyboard': True,
    }
    driver = webdriver.Remote('http://localhost:4723/wd/hub', desired_caps)
    # 单击首页搜索框
    # 延时 20 秒是为了更好地等待系统提示框的出现
    check_and_delay(20)
    resourceId = 'com.taobao.taobao:id/home_searchedit'
    driver.find_element_by_id(resourceId).click()
    check_and_delay()
    # 单击搜索页的搜索框
    text = '玩转 Python 网络爬虫'
    resourceId = 'com.taobao.taobao:id/searchEdit'
    driver.find_element_by_id(resourceId).send_keys(text)
    check_and_delay()
    # 输入搜索内容
    resourceId = 'com.taobao.taobao:id/searchbtn'
    driver.find_element_by_id(resourceId).click()
```

```
    check and delay()
    # 单击销量排序
    sales = 'new UiSelector().description("销量")'
    driver.find element by android uiautomator(sales).click()
    check and delay()
    # 数据写入
    MyList = []
    for t in range(5):
        resourceId = 'com.taobao.taobao:id/auction_layout'
        info = driver.find elements by id(resourceId)
        check and delay()
        for i in info:
            try:
                MyDict = {}
                # 获取标题
                resourceId = 'com.taobao.taobao:id/title'
                title = i.find_element_by_id(resourceId)
                MyDict['title'] = title.text.strip()
                # 获取价格
                resourceId = 'com.taobao.taobao:id/priceBlock'
                price = i.find element by id(resourceId)
                MyDict['price'] = price.get attribute("contentDescription")
                # 去重并写入列表
                if MyDict not in MyList:
                    MyList.append(MyDict)
            except: pass
        # 滑动屏幕
        swipeUp(1000)
    print(MyList)
    # 关闭淘宝 App
    driver.quit()
```

10.7　本 章 小 结

Appium 是一个开源、跨平台的测试框架，可以用来测试原生及混合的移动端应用，支持 iOS、Android 及 FirefoxOS 平台，但利用 Appium 的定位功能也可实现数据爬取。

在 Windows 系统上搭建 Appium 开发环境，需要安装 JavaJDK、Android SDK、Node.JS、Appium-Server 和 Appium-Client，具体的安装说明如下。

- Java JDK：搭建 Java 的开发环境。
- Android SDK：Android 软件开发包，基于 Java 的开发环境运行，可以在计算机启用 Android 模拟器或者连接 Android 手机。
- Node.JS：搭建 Node.JS 的开发环境。
- Appium-Server：安装 Appium 的服务器，基于 Node.JS 的开发环境运行。
- Appium-Client：安装 Appium 的客户端，编写并运行 Appium 自动化代码。

Appium 与 Android 通信连接的代码是相对比较固定的，在连接代码中根据 Android 系统信息

进行相应的修改即可实现连接。Appium 设置了许多连接参数，不同的参数负责实现不同的功能，这些功能主要是对 Android 系统进行设置，以便满足我们的开发需求。

Android 系统的元素查找需要借助软件 UI Automator Viewer 实现，具体操作步骤如下：

步骤01 将手机与计算机进行连接，连接之前确保手机已开启 USB 调试模式。

步骤02 唤醒手机屏幕，当手机界面出现 USB 调试提示信息时，单击“确定”按钮。

步骤03 打开软件 UI Automator Viewer，单击“DeviceScreenshot”按钮捕捉手机当前界面。

步骤04 捕捉成功后，在软件的左侧出现手机界面的截图，在截图里单击某个元素可获取该元素的信息。

Appium 对元素的定位与操作是在 Selenium 的基础上进行实现和扩展，具体的定位与操作方法可以在 Python 安装目录\Lib\site-packages\appium\webdriver\webdriver.py 文件里查阅。

第 11 章

Splash、Mitmproxy 与 Aiohttp

11.1 Splash 动态数据抓取

11.1.1 简介及安装

Splash 是具有 JavaScript 渲染功能并带有 HTTP API 的轻量级浏览器，同时还对接了 Python 的网络引擎框架 Twisted 和 QT 库，让服务具有异步处理能力，以发挥 webkit 的并发能力。简单来说，Splash 是一个带有 API 接口的轻量级浏览器，使用它提供的 API 接口可以简单实现 Ajax 动态数据的抓取。其实它与 Selenium 所实现的功能都是相同的，只不过实现的过程和原理有所不同。

Splash 的安装是基于 Docker 应用容器引擎，Docker 支持三大操作系统：Linux、MacOS 和 Windows。本书以 Windows 安装 Docker 和 Splash 为例，首先下载 DockerToolbox，这是 Docker 的安装包，在浏览器访问 https://docs.docker.com/toolbox/toolbox_install_windows/并单击“Get Docker Toolbox for Windows”即可下载，如图 11-1 所示。



图 11-1　下载 Docker

Docker 的安装包下载后是一个 exe 文件，这是 Windows 的应用程序安装包，直接以管理员身份运行并根据提示安装即可，安装过程的功能选择默认即可。Docker 安装完成后，会在电脑桌面上出现三个图标，如图 11-2 所示。

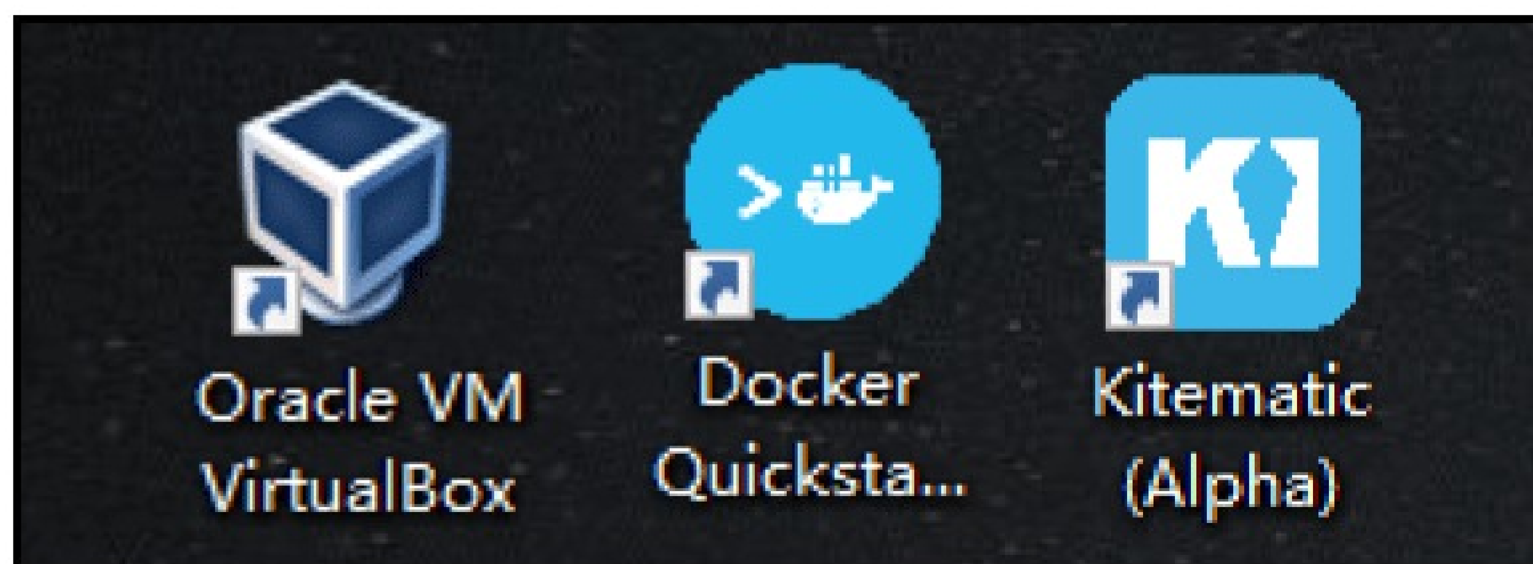


图 11-2　Docker 应用程序

我们单击 Docker Quickstart Terminal 图标，从而打开一个 Docker Toolbox Terminal。首次打开会自动进行一系列配置，直到出现“鲸鱼”的字符画，同时记录 Docker 的 IP 地址：192.168.99.100，如图 11-3 所示。

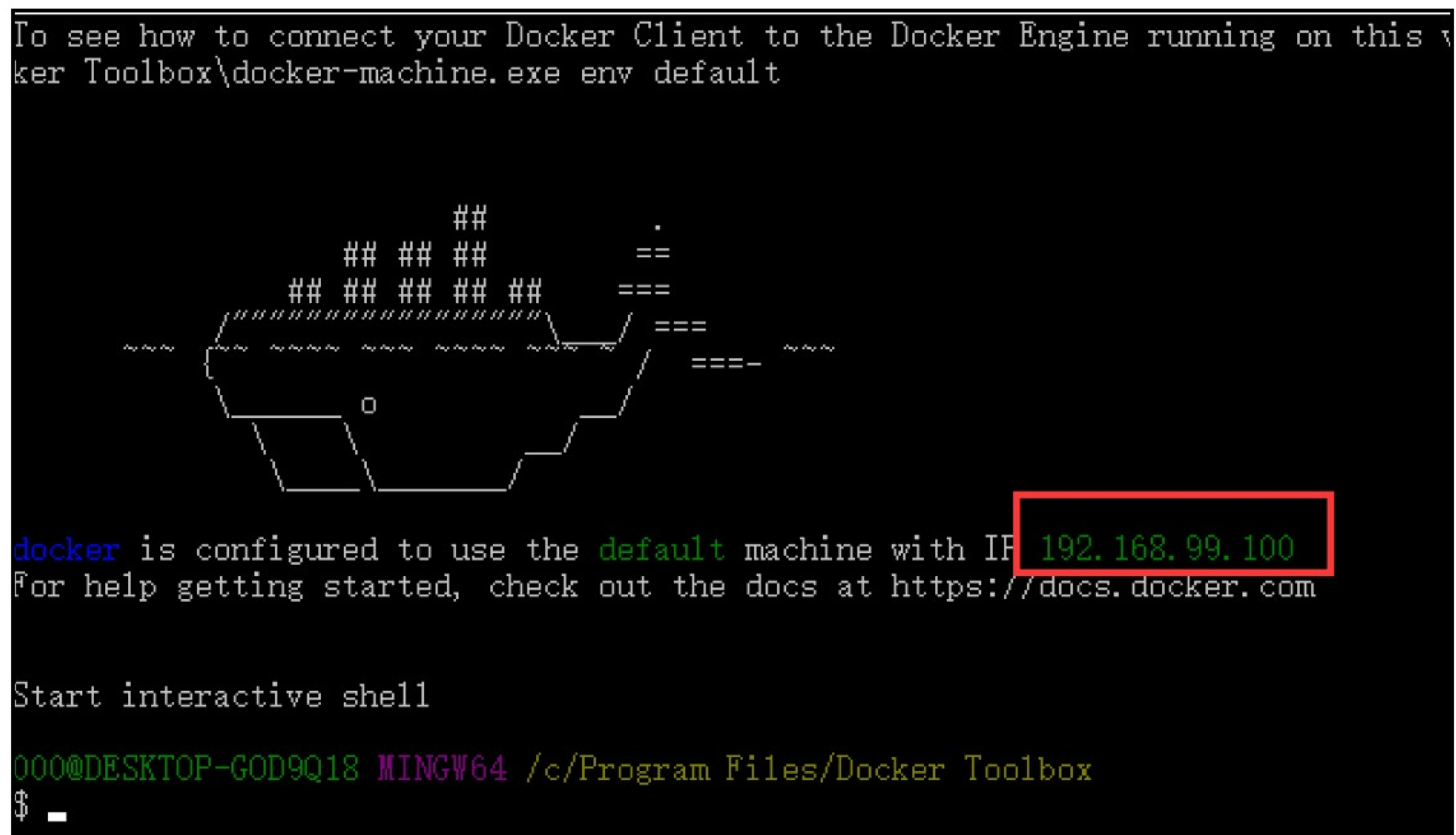


图 11-3　运行 Docker

成功启动 Docker 之后，在图中的“$”后输入各种 Docker 命令就可以使用 Docker。通过 Docker 指令来安装 Splash，输入安装指令 docker run -p 8050:8050 scrapinghub/splash 并按下回车键，等待 Splash 安装即可，直到出现“Starting factory......”即代表 Splash 安装成功，如图 11-4 所示。

```
000@DESKTOP-GOD9Q18 MINGW64 /c/Program Files/Docker Toolbox
$ docker run -p 8050:8050 scrapinghub/splash
2018-11-06 08:53:03+0000 [-] Log opened.
2018-11-06 08:53:03.338854 [-] Splash version: 3.2
2018-11-06 08:53:03.367963 [-] Qt 5.9.1, PyQt 5.9, WebKit 602.1, sip 4.19.3, Twisted 16.1.1, Lua 5.
2018-11-06 08:53:03.369154 [-] Python 3.5.2 (default, Nov 23 2017, 16:37:01) [GCC 5.4.0 20160609]
2018-11-06 08:53:03.370111 [-] Open files limit: 1048576
2018-11-06 08:53:03.371655 [-] Can't bump open files limit
2018-11-06 08:53:03.521281 [-] Xvfb is started: ['Xvfb', ':988708647', '-screen', '0', '1024x768x24
]
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
2018-11-06 08:53:05.534987 [-] proxy profiles support is enabled, proxy profiles path: /etc/splash/
2018-11-06 08:53:06.150941 [-] verbosity=1
2018-11-06 08:53:06.151289 [-] slots=50
2018-11-06 08:53:06.151449 [-] argument_cache_max_entries=500
2018-11-06 08:53:06.152475 [-] Web UI: enabled, Lua: enabled (sandbox: enabled)
2018-11-06 08:53:06.152902 [-] Server listening on 0.0.0.0:8050
2018-11-06 08:53:06.154311 [-] Site starting on 8050
2018-11-06 08:53:06.154784 [-] Starting factory <twisted.web.server.Site object at 0x7ff0c47157f0>
```

图 11-4　安装 Splash

从图 11-4 上可以看到，Splash 的 Server 地址为 0.0.0.0:8050，这个地址是 Docker 的本地 IP 地址，如果我们要在 Windows 中访问 Splash，应将 Docker 的本地地址改为实际的 IP 地址，即图 11-3 的 IP 地址：192.168.99.100。在 Windows 的浏览器访问 http://192.168.99.100:8050/即可打开 Splash，如图 11-5 所示。

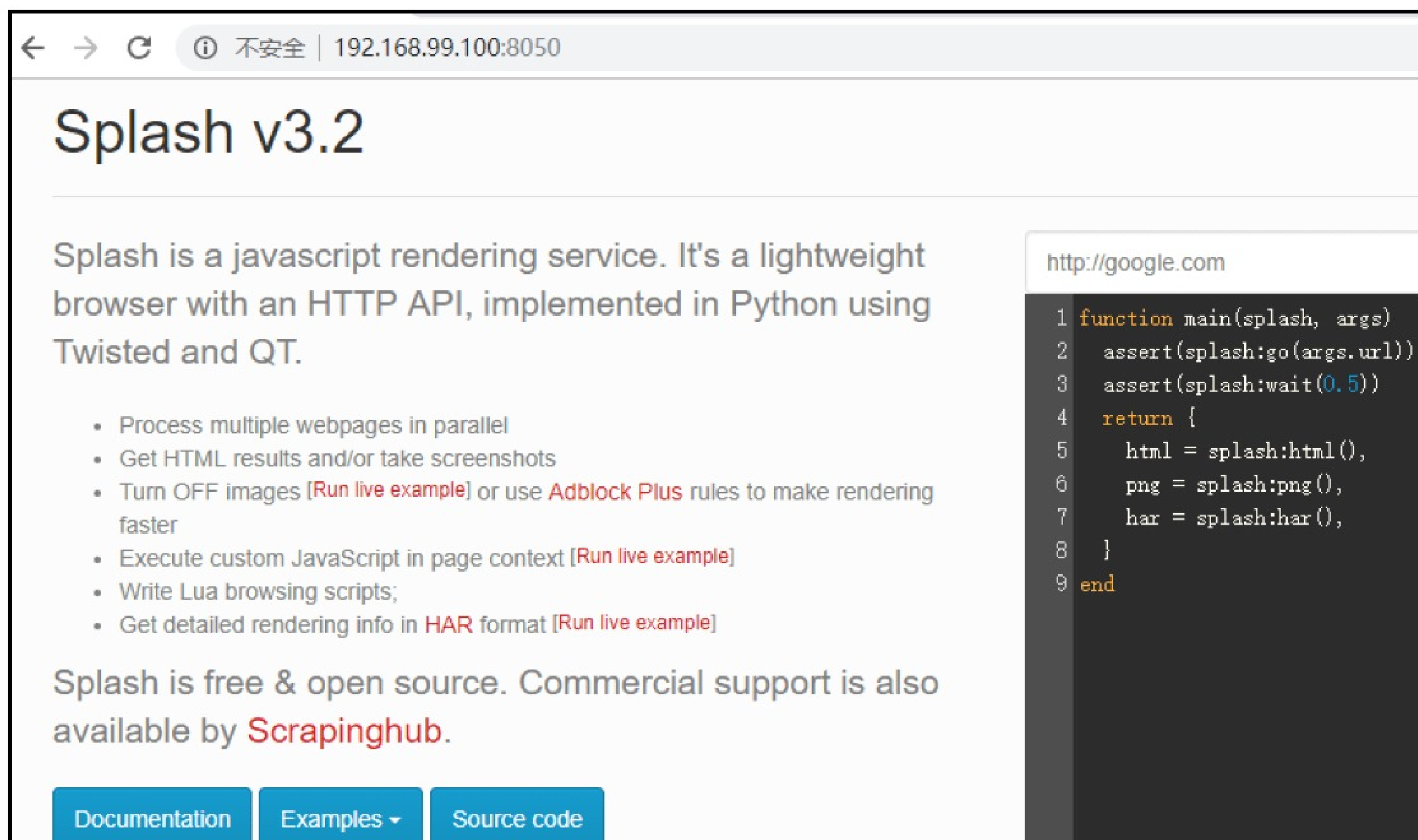


图 11-5　打开 Splash

如果电脑关闭了 Docker Toolbox Terminal 或重启电脑，若想再次运行 Splash，在 Docker Toolbox Terminal 再次输入指令 docker run -p 8050:8050 scrapinghub/splash 即可。

11.1.2 使用 Splash 的 API 接口

Splash 最大的作用是可以执行 JavaScript 代码，将 Ajax 动态数据直接加载到网页上，无需开发者花费时间和精力分析 Ajax 请求，从而实现相关数据的抓取。

Python 可以使用 Splash 提供的 API 接口，从而实现 Python 与 Splash 之间的交互。Splash 提供多种 API 接口实现不同的功能，本书只列出一些较为常用的 API 接口及使用方法，代码如下：

```
import requests
headers = {
    'User-Agent': 'Mozilla/5.0 (X11; Linux x86 64)
                   AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
                   48.0.2564.116 Safari/537.36',
    }
target url = 'https://y.qq.com/portal/singer list.html'

# render.html 获取 JS 加载后的网页信息
url = 'http://192.168.99.100:8050/render.html?
        url='+target url+'&wait=5'
response = requests.get(url, headers=headers)
print(response.text)

# render.png 获取网页截图
url = 'http://192.168.99.100:8050/render.png?
        url='+target url+'&width=500&height=500'
response = requests.get(url, headers=headers)
with open('image.png', 'wb') as f:
    f.write(response.content)

# render.json 返回请求数据
url = 'http://192.168.99.100:8050/render.json?
        url='+target url+'&wait=5'
response = requests.get(url, headers=headers)
print(response.text)

# execute 执行 Lua 脚本
# 因为 Splash 支持 Lua 脚本操作
import requests
from urllib.parse import quote
luaScript = '''
function main(splash)
    return 'Python'
end
'''
# Lua 脚本转码处理
url = 'http://192.168.99.100:8050/execute?
        lua source=' + quote(luaScript)
response = requests.get(url)
print(response.text)
```

从 API 接口的使用方法可以看到，爬取的网站是从 Splash 进行加载，然后再使用 Requests 模块对 Splash 发出请求，从而获取网站的网页内容。在爬虫开发过程中，API 接口 render.html 和 execute

的使用频率相对较高，特别是 execute，它对于复杂的网站也能满足多方面的需求，比如编写 Lua 脚本获取网站的 Cookies 和设置请求头等操作。由于 Splash 的对象方法较多，本节只列出爬虫开发中较为常用的对象方法，具体如下所示：

1. go()方法

该方法对某个链接发送 HTTP 请求，模拟 Get 和 Post 请求，方法定义如下：

```
ok,reason=splash.go{url,baseurl,headers,http_method,body,formdata}
```

【参数解释】

- url：请求的 url。
- baseurl：可选参数，默认空，表示资源加载相对路径。
- headers：可选参数，默认空，表示请求头。
- http_method：可选参数，默认为 GET，支持 POST 请求。
- body：可选参数，默认空，发 POST 请求的表单数据，数据以 JSON 格式表示。
- formdata：可选参数，默认空，发 POST 请求的表单数据，数据以 x-www-form-urlencoded 格式表示。

方法返回变量 ok 和 reason，前者是返回请求结果，后者是返回 HTTP 的状态码。如果 ok 为空则说明网页加载的时候出错，reason 会记录相应的错误信息，反之则说明网页加载成功。

```
# Lua 脚本
function main(splash, args)
  local ok,reason=splash:go{"http://httpbin.org/post",
                    http_method="POST", body="name=Hubo"}
  if ok then
        return splash:html()
  end
end
```

2. wait()方法

该方法用于控制网页加载的等待时间，方法定义如下：

```
splash:wait{time,cancel_on_redirect=false,cancel_on_error=true}
```

【参数解释】

- time：等待的秒数。
- cancel_onredirect：可选参数，默认 false，表示发生重定向就停止等待，并返回重定向结果。
- cancel_on_error：可选参数，默认 false，表示发生了加载错误就停止等待。

```
# Lua 脚本
function main(splash, args)
  splash:go("http://httpbin.org")
  splash:wait(5)
  return {
    html = splash:html()
  }
end
```

3．http_get()方法

该方法用于模拟发送 HTTP 的 GET 请求，方法定义如下：

```
response = splash:http_get{url, headers=nil, follow_redirects=true}
```

【参数解释】

- url：请求的 url 地址。
- headers：可选参数，默认空，设置请求头。
- follow_redirects：可选参数，是否启动自动重定向。

```
# Lua 脚本
function main(splash, args)
  local treat = require("treat")
  local response = splash:http_get("http://httpbin.org/get")
  return {
    html = treat.as_string(response.body),
    url = response.url,
    statud = response.status
  }
end
```

4．http_post()方法

该方法模拟发送 HTTP 的 POST 请求，方法定义如下：

```
response = splash: http_post{url, headers=nil, follow_redirects=true,
               body=nil}
```

【参数解释】

- url：请求的 url 读取。
- headers：可选参数，默认空，设置请求头。
- follow_redirects：可选参数，是否启动自动重定向。
- body：可选参数，默认空。

```
# Lua 脚本
function main(splash, args)
  local treat = require("treat")
  local response = splash:http_post("http://httpbin.org", body="name=Python")
  return {
    html = treat.as_string(response.body),
    url = response.url,
    statud = response.status
  }
end
```

5．get_cookies()方法

该方法用于获取当前页面的 Cookies，使用方法如下：

```
# Lua 脚本
function main(splash)
  splash:go("https://www.baidu.com")
```

```
    return {
      # 返回 Cookies 信息
      splash:get cookies()
    }
  end
```

6. add_cookies()方法

该方法用于为当前页面添加 Cookies，定义方法如下：

```
cookies = splash:add cookie{name, value, path=nil, domain=nil,
                    expires=nil, httpOnly=nil, secure=nil}
```

【参数解释】

- name：当前 Cookies 的名称。
- value：当前 Cookies 的数值。
- path：可选参数，默认空，代表 Cookies 所在的目录。
- domain：可选参数，默认空，代表 Cookies 所在的域。
- expires：可选参数，默认空，代表 Cookies 的有效期。
- httpOnly：可选参数，默认空，设置 Cookies 是否支持 JS 读取。
- secure：可选参数，默认空，设置 Cookies 的跨域传递等安全问题。

```
# Lua 脚本
function main(splash)
    # 如有多条 Cookies，则多次使用 add_cookies()添加
    splash:add cookie{"sessionid", "12346"}
    splash:go("https://www.baidu.com/")
    return splash:html()
end
```

7. set_custom_headers()方法

该方法用于设置请求头，使用方法如下：

```
# Lua 脚本
function main(splash)
  splash:set custom headers({
      ["User-Agent"] = "Splash",
      ["Referer"] = "https://www.baidu.com/"
    })
  splash:go("http://httpbin.org/get")
  return {
    splash:html()
  }
end
```

上述 7 个对象方法是网络爬虫中最为常用的方法，此外，Splash 还定义了很多对象方法，有兴趣的读者可以参考 Splash 的官方网站（https://splash.readthedocs.io/en/latest/）。

11.2 Mitmproxy 抓包

11.2.1 简介及安装

Mitmproxy 是一个支持 HTTP 和 HTTPS 的抓包程序，实现的功能与 Fiddler 抓包工具相同，只不过它是以控制台的形式操作。Mitmproxy 还有两个关联组件：Mitmweb 与 Mitmdump，前者是一个基于 Web 的 Mitmproxy 接口，它可以清楚地观察 Mitmproxy 捕获的请求与响应；后者是 Mitmproxy 的命令行接口，它可以帮助我们对接 Python 脚本，使用 Python 实现监听后的处理。Mitmproxy 还具备以下功能：

（1）拦截 HTTP 和 HTTPS 请求和响应，并允许动态修改请求和响应。

（2）保存完整的 HTTP 会话，以便重播和分析。

（3）可重播 HTTP 会话的客户端（即手机端）。

（4）记录所有的 HTTP 响应。

（5）反向代理模式，用于将流量转发到指定的服务器。

（6）MacOS 和 Linux 系统具有透明代理模式。

（7）使用 Python 对 HTTP 流量进行脚本操作。

（8）用于拦截的 SSL/TLS 证书是即时生成。

总的来说，Mitmproxy 的功能十分强大，而在爬虫开发中，它可以帮助我们对接 Python 脚本，就这一功能已经能帮助我们解决爬虫中常见的问题，比如爬取手机 App 应用的图片、视频、文字内容等。Appium 侧重于手机的自动化操控，对 App 的图片、视频、文字内容爬取相对困难，而 Mitmproxy 则补完了 Appium 的缺点，可以说 Appium+Mitmproxy 是手机 App 爬虫开发的利器。

Mitmproxy 支持三大操作系统：Linux、MacOS 和 Windows。本书以 Windows 系统为例，在 CMD 窗口下输入 Mitmproxy 的安装指令 pip install mitmproxy，即可完成 Mitmproxy 的安装。

11.2.2 用 Mitmdump 抓取爱奇艺视频

完成 Mitmproxy 的安装后，下面开始介绍 Mitmproxy 的使用。首先我们将一台安卓系统的手机与电脑处于同一个局域网内，最好手机与电脑都是连接同一个无线网络。在电脑上打开 CMD 窗口，输入 ipconfig 查看当前电脑的无线网络 IP 地址，如图 11-6 所示。

根据电脑的无线网络 IP 地址设置手机的 WLAN 代理，以华为手机为例，打开“设置”->“无线和网络”->“WLAN”，长按当前已连接的无线网络并选择“修改网络”，单击“显示高级选项”，将代理设置为手动，并在“服务器主机名”和“服务器端口”分别输入“192.168.1.102”和“8080”，如图 11-7 所示。

```
无线局域网适配器 WLAN:

   连接特定的 DNS 后缀 . . . . . . . :
   本地链接 IPv6 地址. . . . . . . . : fe80::e0ee:5544:a20:a16d%9
   IPv4 地址 . . . . . . . . . . . . : 192.168.1.102
   子网掩码  . . . . . . . . . . . . : 255.255.255.0
   默认网关. . . . . . . . . . . . . : 192.168.1.253
```

图 11-6　电脑的无线网络 IP 地址



图 11-7　设置手机的 WLAN 代理

然后在电脑上打开 CMD 窗口，输入“Mitmweb”并按回车，Mitmproxy 就会运行 Mitmweb 组件并自动弹出相关的网页。当我们在手机浏览器上打开相关的网页，Mitmweb 就会自动捕捉手机上的请求信息与响应内容，如图 11-8 所示。

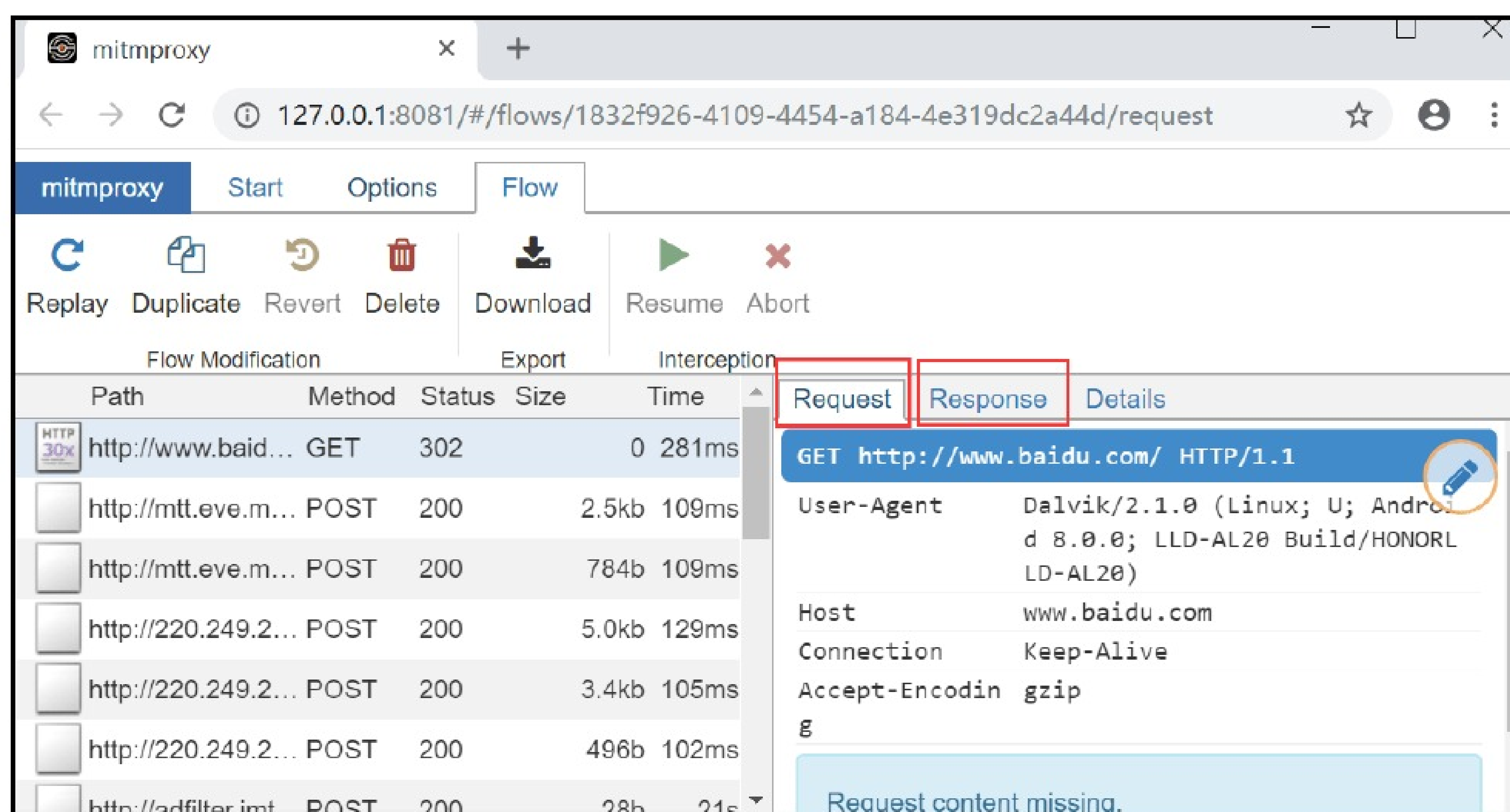


图 11-8　Mitmweb 的抓包信息

通过手机的 WLAN 代理实现手机与 Mitmproxy 连接，下面我们开始使用 Mitmweb 抓取爱奇艺的视频信息。在手机浏览器上输入爱奇艺的网址，以搞笑视频为例（http://m.iqiyi.com/fun/）。单击并播放某个视频，在电脑的 Mitmweb 网页上，将请求信息以“Size”进行降序排列，第一条数据就是当前视频的请求信息，如图 11-9 所示。

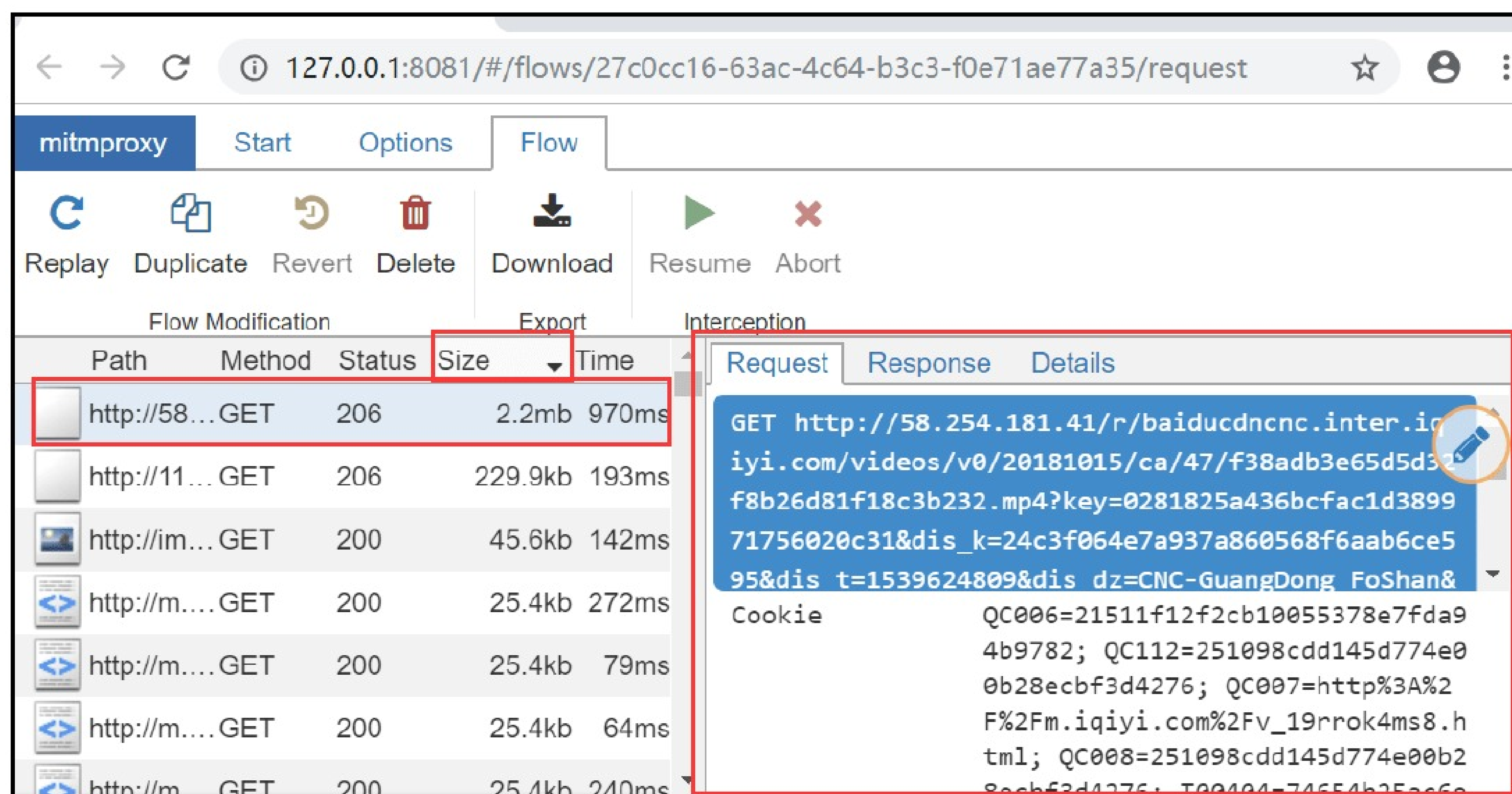


图 11-9　当前视频的请求信息

分析视频的 URL 发现，URL 地址必定含有“/r/baiducdncnc.inter.iqiyi.com/videos/v0/”的内容。读者不妨尝试播放其他视频，观察它们的 URL 地址是否具有这一特征。那么，只要 URL 符合这一特征，我们都可以将其进行视频下载处理。

视频的下载处理由 Mitmproxy 的 Mitmdump 和 Python 共同完成。首先我们创建一个 download.py 文件和一个 video 文件夹，然后在 download.py 文件里定义一个函数 response，函数参数为 flow，代码如下：

```
import requests
# 文件路径
path = r'F:\\video\\'
num = 0
def response(flow):
    global num
    # 视频 URL 的特征
    target_urls = r'/r/baiducdncnc.inter.iqiyi.com/videos/v0/'
    # 过滤重复的 URL
    repeat_urls = []
    if target_urls in flow.request.url and flow.request.url not in repeat_urls:
        repeat_urls.append(flow.request.url)
        # 设置视频名
        filename = path + str(num) + '.mp4'
        # 使用 request 获取视频 URL 的内容
        # stream=True 作用是推迟下载响应体直到访问 Response.content 属性
        res = requests.get(flow.request.url, stream=True)
        # 将视频写入文件夹
```

```
            with open(filename, 'ab') as f:
                f.write(res.content)
                f.flush()
                print(filename + '下载完成')
                num += 1
```

当我们运行 Mitmdump 的时候，Mitmdump 会自动捕捉手机的 HTTP 请求，然后将捕捉到的 HTTP 请求进行处理，调用 download.py 的自定义函数 response，函数参数 flow 代表当前的请求信息。运行 Mitmdump 之前，记得先把 Mitmweb 关闭，否则两者会出现冲突。在 CMD 窗口下，将当前路径切换到 download.py 所在的路径，并输入 Mitmdump 指令即可启动，如下所示

```
F:\>Mitmdump -s download.py
Loading script download.py
Proxy server listening at http://*:8080
```

Mitmdump 运行成功后，在手机上单击并播放某个视频，当前播放的视频就会自动下载并保存到 video 文件夹，如图 11-10 所示。

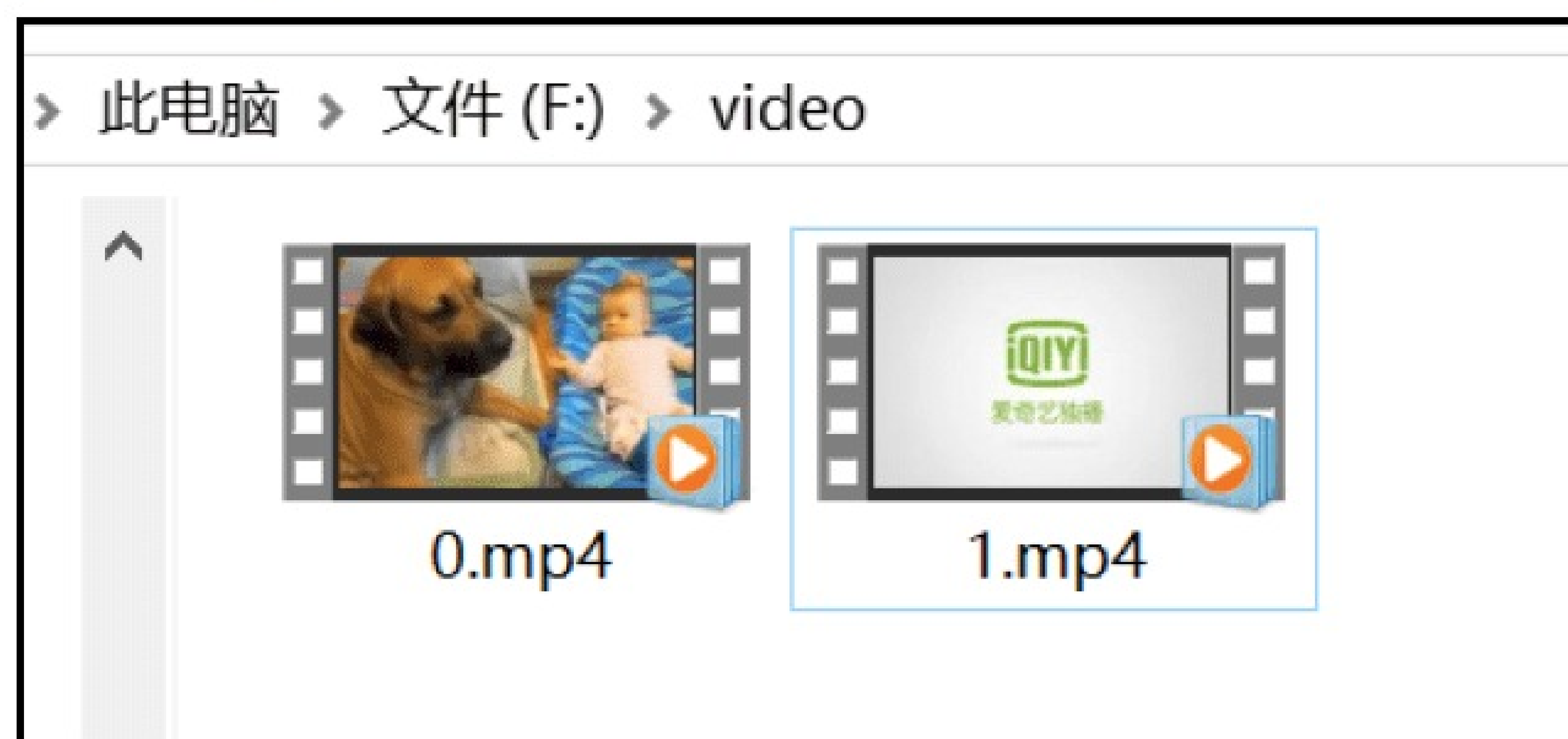


图 11-10　视频下载

上述代码只是定义了函数 response，这是用于处理手机的 HTTP 请求。此外，还可以自定义函数 request，该函数是自定义手机的 HTTP 请求，使其能满足开发需求。一般情况下，默认的 HTTP 请求已能满足大部分的开发需求。

本节只讲述了 Mitmdump 的使用，并没有结合 Appium 的使用，读者可以结合前文 Appium 的内容，进一步完善本节所实现的功能。

11.3　Aiohttp 高并发抓取

11.3.1　简介及使用

Aiohttp 是 Python 的一个第三方网络编程模块，它可以开发服务端和客户端，服务端也就是我们常说的网站服务器；客户端是访问网站的 API 接口，常用于接口测试，也可用于开发网络爬虫。Aiohttp 是基于 Asyncio 实现的 HTTP 框架，Asyncio 是从 Python 3.4 开始引入的标准库，它是因协程的概念而生，这是 Python 官网推荐高并发的模块之一。

由于 Asyncio 具有高并发的特性，因此 Aiohttp 继承了 Asyncio 的特性，使得 Aiohttp 非常适合开发网络爬虫。在使用 Aiohttp 之前，需要安装 Aiohttp 模块，安装方式可以使用 pip 指令完成，也

可以自行下载 whl 安装包（https://www.lfd.uci.edu/~gohlke/pythonlibs/#aiohttp），安装指令如下所示：

```
# pip 在线安装 aiohttp
pip install aiohttp
# 从 whl 安装包安装 aiohttp
pip install aiohttp-3.4.4-cp37-cp37m-win_amd64.whl
```

Aiohttp 模块安装完成后，在 CMD 窗口进入 Python 的交互模式，导入 Aiohttp 模块并验证模块安装是否成功，验证代码如下：

```
C:\Users\000>python
>>> import aiohttp
>>> aiohttp.__version__
'3.4.4'
```

本节中，我们只介绍如何使用 Aiohttp 的客户端功能，通过客户端功能去实现网络爬虫的开发。首先要建立一个会话对象 session，然后利用会话对象 session 去访问网页，基本用法如下：

```
import aiohttp
import asyncio
async def hello(URL):
    async with aiohttp.ClientSession() as session:
        async with session.get(URL) as response:
            response = await response.text()
            print(response)
if __name__ == '__main__':
    URL = 'http://python.org'
    loop = asyncio.get_event_loop()
    loop.run_until_complete(hello(URL))
```

上述例子是使用 Aiohttp 和 Asyncio 模块访问 Python 官方网站。函数 hello()加入了 Python 内置的关键词 async 和 await，这是将函数设置为异步操作，这是 Aiohttp 的使用方式；而函数 hello()的调用和运行需要借助 Asyncio 模块，Aiohttp 只实现网站的访问方式，而代码的执行过程则由 Asyncio 模块实现。

Aiohttp 在发送 HTTP 请求的时候，还可以设置 HTTP 请求的请求头、超时、Cookies 和代理 IP。请求头和代理 IP 是在发送 HTTP 请求的过程中分别设置参数 headers 和 proxy，而超时和 Cookies 是在会话对象 session 里分别设置参数 cookies 和 timeout 实现。具体的设置方法如下：

```
from aiohttp import ClientSession
import aiohttp
URL = 'http://httpbin.org'
# 设置请求头
headers = {'content-type': 'application/json'}
async with ClientSession() as session:
    async with session.get(URL,headers=headers) as response:
        response = await response.text()

# 设置超时，在会话中设置超时
timeout = aiohttp.ClientTimeout(total=60)
async with ClientSession(timeout=timeout) as session:
    async with session.get(URL) as response:
```

```
            response = await response.text()
            print(response)
    # 设置超时，在请求中设置超时
    async with ClientSession() as session:
        async with session.get(URL,timeout=timeout) as response:
            response = await response.text()
            print(response)

    # 设置 Cookies
    cookies = {'cookies': 'working'}
    async with ClientSession(cookies=cookies) as session:
        async with session.get(URL) as response:
            response = await response.text()
            print(response)

    # 设置代理 IP
    proxy = 'http://117.191.11.72:8080'
    async with ClientSession() as session:
        async with session.get(URL,proxy=proxy) as response:
            response = await response.text()
            print(response)
    # 支持代理授权
    async with ClientSession() as session:
        proxy auth = aiohttp.BasicAuth('user', 'pass')
        async with session.get("http://python.org",
                              proxy="http://proxy.com",
                              proxy auth=proxy auth) as resp:
            response = await response.text()
            print(response)
```

Aiohttp 定义了多种 HTTP 请求方法，如 GET、OPTIONS、HEAD、POST、PUT、PATCH 和 DELETE 方法。在网络爬虫中，使用最为频繁的是 GET 和 POST 方法。GET 请求有两种形式，分别是不带参数和带参数，使用方法如下：

```
    # 不带参数
    URL = 'http://httpbin.org/get'
    async with ClientSession() as session:
        async with session.get(URL) as response:
            response = await response.text()
            print(response)

    # 带参数
    # 在 URL 设置参数
    URL = 'http://httpbin.org/get?key=python'
    async with ClientSession() as session:
        async with session.get(URL) as response:
            response = await response.text()
            print(response)

    # 设置请求参数 params
    URL = 'http://httpbin.org/get'
    params = {'wd': 'python'}
    async with ClientSession() as session:
        async with session.get(URL,params=params) as response:
```

```
        response = await response.text()
        print(response)
```

一般情况下，发送 POST 请求都会带有请求参数，参数值会被包含在请求体中，然后一并发到网站服务器，参数值的数据格式可以为字典、JSON、字符串和字节流，不同的数据格式实现不同的功能，使用方式如下：

```
# 以字典格式写入
URL = 'http://httpbin.org/post'
data = {'key': 'python'}
async with ClientSession() as session:
    async with session.post(URL,data=data) as response:
        response = await response.text()
        print(response)

# 以 JSON 格式写入
URL = 'http://httpbin.org/post'
data = {'key': 'python'}
async with ClientSession() as session:
    async with session.post(URL,json=data) as response:
        response = await response.text()
        print(response)

# 以字符串格式写入
URL = 'http://httpbin.org/post'
data = 'python'
async with ClientSession() as session:
    async with session.post(URL,data=data) as response:
        response = await response.text()
        print(response)

# 以字节流格式写入（上存文件）
URL = 'http://httpbin.org/post'
data = {'file': open('log.txt', 'rb')}
async with ClientSession() as session:
    async with session.post(URL,data=data) as response:
        response = await response.text()
        print(response)
```

不管是 GET 还是 POST 请求，最终都要从请求中获取相应的响应内容，从上述例子看到，响应内容可以使用 text()方法获取。除此之外，Aiohttp 还定义了多种获取响应内容的方法，如下所示：

```
# 设置编码格式
response = await response.text(encoding='utf-8')
# 以字节流格式返回
response = await response.read()
# 以 JSON 格式返回
response = await response.json()
# 获取响应状态码
status=response.status
# 获取响应的请求头
headers=response.headers
# 获取 URL 地址
url=response.url
```

综上，Aiohttp 的使用方法与 Requests 有相似之处，而且能轻松实现高并发爬虫开发。有关 Aiohttp 的使用，读者可以查看官方文档（https://aiohttp.readthedocs.io/en/stable/client.html）。

11.3.2　Aiohttp 异步爬取小说排行榜

在本章节中，我们通过一个简单的案例来深入了解 Aiohttp 的应用，案例实现过程如下：

（1）爬取对象是起点小说网的 24 小时热销榜。

（2）数据清洗会使用第三方模块 BeautifulSoup4 实现。

（3）数据将以 CSV 文件进行存储。

数据清洗和数据存储的知识要点会在后续的章节详细讲述，本章只做简单的介绍。由于项目需要使用 BeautifulSoup4 模块，因此在开发环境中安装该模块，安装方式可使用 pip 在线安装即可，打开 CMD 窗口并输入安装指令：

```
pip install beautifulsoup4
```

下一步分析 24 小时热销榜的网页结构，在浏览器访问网址（www.qidian.com/rank/hotsales?page=1）并打开浏览器的开发者工具，点击 Network 选项卡的 Doc 标签，如图 11-11 所示。



图 11-11　网页结构

分析网页结构是要根据爬虫的爬取方式而决定，爬取方式主要分为两类，说明如下：

（1）如果使用 Selenium 或 Splash 爬取数据，网页分析需要在开发者工具的 Elements 选项卡里进行，因为 Selenium 和 Splash 是获取网页加载后的内容。

（2）如果使用 Requests 或 Aiohttp 这类模块去爬取数据，则由开发者工具的 Network 选项卡进行网页分析，并且还要在各个分类标签里找到数据所对应的请求方式。

从图上看到，网页上的小说信息可以在 Doc 标签里找到对应的 HTML 源码，并且 Doc 标签的请求地址与浏览器的地址栏是一致的，也就是说，我们只需对网页地址发送 HTTP 请求即可获取小说信息。

在网页最下方设有分页功能，当点击不同的页数按钮，浏览器地址栏的 URL 地址会随之变化。如第一页的 page=1、第二页的 page=2、第三页的 page=3……以此类推，参数 page 代表分页功能的页数，URL 地址根据页数的不同来显示相应的小说信息，如图 11-12 所示。



图 11-12　URL 变化规律

从上述的分析得知，只要动态改变 URL 地址的参数 page 即可得到不同页数的网页内容，然后将网页内容进行数据清洗并提取相应的小说信息，最后将小说信息写入 CSV 文件。因此，项目的功能代码如下所示：

```
import asyncio
from aiohttp import ClientSession
# 导入数据清洗库 BeautifulSoup
from bs4 import BeautifulSoup
# 导入内置的 CSV 库
import csv

# 定义网站访问函数 getData，将网站内容返回
async def getData(url, headers):
    #  创建会话对象 session
    async with ClientSession() as session:
        # 发送 GET 请求，并设置请求头
        async with session.get(url,headers=headers) as response:
            # 返回响应内容
            return await response.text()

def savaData(result):
    for i in result:
        soup = BeautifulSoup(i, 'html.parser')
        find div = soup.find all('div',class ='book-mid-info')
        for d in find div:
            name = d.find('h4').getText()
            author = d.find('a',class ='name').getText()
            update = d.find('p',class ='update').getText()
            # 写入 CSV 文件
            csvFile = open('data.csv','a', newline='')
            writer = csv.writer(csvFile)
```

```
            writer.writerow([name,author,update])
            csvFile.close()

# 定义运行函数 run
def run():
    for i in range(25):
        # 构建不同的 URL 地址并传入函数 getData，最后由 asyncio 模块执行
        task=asyncio.ensure future(getData(url.format(i+1),headers))
        # 将所有请求加入到列表 tasks
        tasks.append(task)
    # 等待所有请求执行完成，一并返回全部的响应内容
    result = loop.run until complete(asyncio.gather(*tasks))
    savaData(result)
    print(len(result))

if   name    == '  main  ':
    headers = {
        'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                       AppleWebKit/537.36 (KHTML, like Gecko)
                       Chrome/69.0.3497.100 Safari/537.36'
    }
    tasks = []
    url = "https://www.qidian.com/rank/hotsales?page={}"
    # 创建 get event loop 对象
    loop = asyncio.get event loop()
    # 调用函数 run
    run()
```

上述代码中，一共定义了三个函数和运行函数__main__，各个函数实现的功能说明如下：

（1）getData()是使用 Aiohttp 模块发送 HTTP 请求，参数 url 和 headers 分别代表请求地址和请求头，函数将响应内容作为返回值。

（2）savaData()是将响应内容进行数据清洗处理，从数据中提取小说信息并写入 CSV 文件，参数 result 代表排行榜所有分页的网页内容。

（3）run()是遍历 25 次来构建不同的 URL 地址，每次遍历是由 Asyncio 调用函数 getData()，然后传入当前的URL地址，生成任务对象task，每次遍历所生成的任务对象task都会写入列表tasks，最后由 Asyncio 调度执行任务列表，将全部的执行结果（响应内容）以列表返回，赋值给变量 result。

（4）运行函数__main__是定义请求头、格式化 URL 地址、创建 get_event_loop 对象和调用函数 run()，这是为函数 getData()和 run()的变量进行初始化处理。

我们在 PyCharm 里运行上述代码，排行榜 25 个分页所爬取的时间约 3 秒左右，这样的爬取效率归功于 Aiohttp 的异步并发特性。最后打开 data.csv 文件并查看排行榜的小说信息，如图 11-13 所示。

	A	B	C
1	全球高武	老鹰吃小鸡	最新更新 第555章 不够理智·2018-11-26 17:37
2	天道图书馆	横扫天涯	最新更新 第一千六百五十七章 天涯让贤【第三更，第58294加更】·2018-11-26 12:00
3	诡秘之主	爱潜水的乌贼	最新更新 第十二章 舌尖上的鱼人（周一求月票推荐票）·2018-11-26 12:35
4	明朝败家子	上山打老虎额	最新更新 第七百九十四章：忠义之名·2018-11-26 13:24
5	大王饶命	会说话的肘子	最新更新 很简单的完本感言·2018-11-26 07:34
6	我有一座恐怖屋	我会修空调	最新更新 第362章 你在哪？·2018-11-26 17:58
7	牧神记	宅猪	最新更新 第一千一百二十四章 骨头·2018-11-26 12:00
8	将夜	猫腻	最新更新 汇报三件事情·2014-05-04 21:57
9	民国谍影	寻青藤	最新更新 第三百五十四章 确定目标（求月票）·2018-11-26 09:56
10	凡人修仙之仙界篇	忘语	最新更新 第五百七十四章 濒临极限·2018-11-26 18:34
11	道君	跃千愁	最新更新 第一二三五章 愿娶·2018-11-26 16:50
12	史上最强赘婿	沉默的糕点	最新更新 第177章:肥宅逆天!仇枭战栗!养肥了杀!(1更)·2018-11-26 13:07
13	修真聊天群	圣骑士的传说	最新更新 第2347章 上天无路，入地无门·2018-11-26 00:21
14	汉乡	孑与2	最新更新 第九十八章暖心暖肺的阿娇·2018-11-25 15:10
15	学霸的黑科技系统	晨星LL	最新更新 第576章 意料之外的蛋糕？·2018-11-26 10:00
16	大医凌然	志鸟村	最新更新 第391章 滚滚向前·2018-11-26 12:43
17	圣墟	辰东	最新更新 第1310章 万物母气·2018-11-24 23:23
18	超神机械师	齐佩甲	最新更新 739 来客·2018-11-26 18:11

图 11-13　小说信息

虽然 Aiohttp 的异步并发可以提高爬虫的爬取效率，但也会因为爬取速度过快而被网站判为爬虫机器人，从而引发一系列的反爬虫机制。

11.4 本章小结

（1）Splash 是具有 JavaScript 渲染功能并带有 HTTP API 的轻量级浏览器，同时还对接了 Python 的网络引擎框架 Twisted 和 QT 库。简单来说，Splash 是一个带有 API 接口的轻量级浏览器，通过使用它提供的 API 接口可以简单实现 Ajax 动态数据的抓取。

Splash 最大的作用是可以执行 JavaScript 代码，将 Ajax 动态数据直接加载到网页上，无需开发者花费时间和精力分析 Ajax 请求，从而实现相关数据的抓取。

Python 可以使用 Splash 提供的 API 接口，从而实现 Python 与 Splash 之间的交互。Splash 提供多种 API 接口实现不同的功能，本书只列出了一些较为常用的 API 接口及使用方法。

（2）Mitmproxy 是一个支持 HTTP 和 HTTPS 的抓包程序，实现的功能与 Fiddler 抓包工具相同，只不过它是以控制台的形式操作。Mitmproxy 还有两个关联组件：Mitmweb 与 Mitmdump，前者是一个基于 Web 的 Mitmproxy 接口，它可以清楚地观察 Mitmproxy 捕获的请求与响应；后者是 Mitmproxy 的命令行接口，它可以帮助我们对接 Python 脚本，使用 Python 实现监听后的处理。Mitmproxy 还具备以下功能：

- 拦截 HTTP 和 HTTPS 请求和响应，并允许动态修改请求和响应。
- 保存完整的 HTTP 会话，以便重播和分析。
- 可重播 HTTP 会话的客户端（即手机端）。
- 记录所有 HTTP 响应。
- 反向代理模式，用于将流量转发到指定的服务器。
- MacOS 和 Linux 系统具有透明代理模式。
- 使用 Python 对 HTTP 流量进行脚本操作。

- 用于拦截的 SSL/TLS 证书是即时生成。

总的来说，Mitmproxy 的功能是十分强大，而在爬虫开发中，它可以帮助我们对接 Python 脚本，就这一功能已经能帮助我们解决爬虫中常见的问题，比如爬取手机 App 应用的图片、视频、文字内容等。Appium 侧重于手机的自动化操控，对 App 的图片、视频、文字内容爬取相对困难，而 Mitmproxy 则补充了 Appium 的缺点，可以说 Appium+Mitmproxy 是手机 App 爬虫开发的利器。

（3）Aiohttp 是 Python 的一个第三方网络编程模块，它可以开发服务端和客户端，服务端也就说我们常说的网站服务器；客户端是访问网站的 API 接口，常用于接口测试，也可用于开发网络爬虫。Aiohttp 是基于 Asyncio 实现的 HTTP 框架，Asyncio 是从 Python 3.4 开始引入的标准库，它是因为协程的概念而生，这是 Python 官网推荐高并发的模块之一。

Aiohttp 的使用方法与 Requests 模块具有相似之处，而且 Aiohttp 具有异步并发的特性，可以轻松实现高并发的爬虫开发。

第 12 章

验证码识别

12.1 验证码的类型

在开发爬虫时，经常会遇到验证码识别，在网站中加入验证码的目的是加强用户安全性和提高反爬虫机制，有效防止对某一个特定注册用户用特定程序暴力破解的方式不断地进行登录尝试。在此简单地为大家介绍一下验证码的种类。

- 字符验证码：在图片上随机产生数字、英文字母或汉字，一般有 4 位或者 6 位验证码字符。通过添加干扰线、添加噪点以及增加字符的粘连程度和旋转角度来增加机器识别的难度。但是这种传统的验证码随着 OCR 技术的发展，能够轻易地被破解。
- 图片验证码：图片验证码也只是换汤不换药，应用了字符验证码的技术，只是不是随机的字符，而是让人识别图片，比如 12306 的验证码。同时还包括一些将广告嵌入图片上面的验证码，都应该归属到这一类。
- GIF 动画验证码：主流验证码都提供的是静态的图片，比较容易被 OCR 软件识别，有的网站提供 GIF 动态的验证码图片，使得识别器不容易辨识哪一个图层是真正的验证码图片，在提供清晰图片的同时，可以更有效地防止识别器的识别。据统计，动画 GIF 验证码的防垃圾注入可以达到 100%，是一个非常有效的验证码创新模式。同时，GIF 动画效果多达百种，也可以增加网站页面的美观效果。
- 极验验证码：这是极验验证于 2012 年推出的新型验证码，基于行为式验证技术，通过拖动滑块完成拼图的形式实现验证，是目前看到的比较有创意的验证码，安全性具有新的突破。
- 手机验证码：通过短信的形式发送到用户手机上面的验证码，一般为 6 位的数字。
- 语音验证码：也属于手机端验证的一种方式。
- 视频验证码：视频验证码是验证码中的新秀，在视频验证码中，将随机数字、字母和中文

组合而成的验证码动态嵌入 MP4、FLV 等格式的视频中，增大破解难度。验证码视频动态变换、随机响应，可以有效地防范字典攻击、穷举攻击等攻击行为。视频中的验证码字母、数字组合，字体的形状、大小，速度的快慢，显示效果和轨迹的动态变换，增加了恶意抓屏破解的难度。其安全度远高于普通的验证码，而且这种验证码形式使用户不会感到枯燥，由于其提高了机器识别的难度，因此可以降低用户识别的难度，使得用户更容易辨认。

现在大多数网站还使用字符验证码，主要用于用户登录。有些网站数据需要用户登录才有访问权限，爬取这样的数据时，首先要完成用户登录，获取访问权限才能继续下一步的数据爬取。

对于用户登录设置验证码识别的网站有三种解决方案：

（1）人工识别验证码。将验证码图片下载到本地，然后靠使用者自行识别并将识别内容输入，程序获取输入内容，完成用户登录。其特点是开发简单，适合初学者，但过分依赖人为控制，难以实现批量爬取。

（2）通过 Python 调用 OCR 引擎识别验证码。这是最理想的解决方案，但正常情况下，OCR 准确率较低，需要机器学习不断提高 OCR 准确率，开发成本相对较高。

（3）调用 API 使用第三方平台识别验证码。开发成本较低，有完善的 API 接口，直接调用即可，识别准确率高，但每次识别需收取小额费用。

上述方案是目前解决验证码最有效的手段，本章主要介绍如何使用 OCR 技术和第三方平台识别验证码。

12.2　OCR 技术

OCR（Optical Character Recognition，光学字符识别）是指电子设备（例如扫描仪或数码相机）检查纸上打印的字符，通过检测暗、亮的模式确定其形状，然后用字符识别方法将形状翻译成计算机文字的过程，即针对印刷体字符，采用光学的方式将纸质文档中的文字转换成为黑白点阵的图像文件，并通过识别软件将图像中的文字转换成文本格式，供文字处理软件进一步编辑加工的技术。

在 Python 中，支持 ORC 的模块有 pytesser3 和 pyocr，其原理主要是通过模块功能调用 OCR 引擎识别图片，OCR 引擎再将识别的结果返回到程序中，本节主要以 pyocr 为例进行介绍。在 Windows 中安装 pyocr 可以在 CMD 下使用 pip 安装：

```
pip install pyocr
```

安装 pyocr 模块之后，还需要安装 PIL 模块，这是专门用于处理图片的模块，pyocr 依赖该模块才能完成识别，pip 安装如下：

```
pip install Pillow
```

完成 pyocr 和 PIL 模块的安装后，最后是 OCR 引擎的安装，图像识别主要由 OCR 引擎完成，pyocr 只起到一个调用引擎的作用。

Tesseract-OCR 是一个免费、开源的 OCR 引擎，读者可从网上自行搜索下载安装。在 Windows 系统中，OCR 引擎（Tesseract-OCR）可通过.exe 安装包安装。值得注意的是，在安装过程中有附

加功能选项，如图 12-1 所示。

选项“Tesseract development files”是 OCR 开发文件，可以在这个引擎的基础上进行二次开发。若安装时勾选该选项，则安装过程中会访问谷歌服务器下载开发文件。

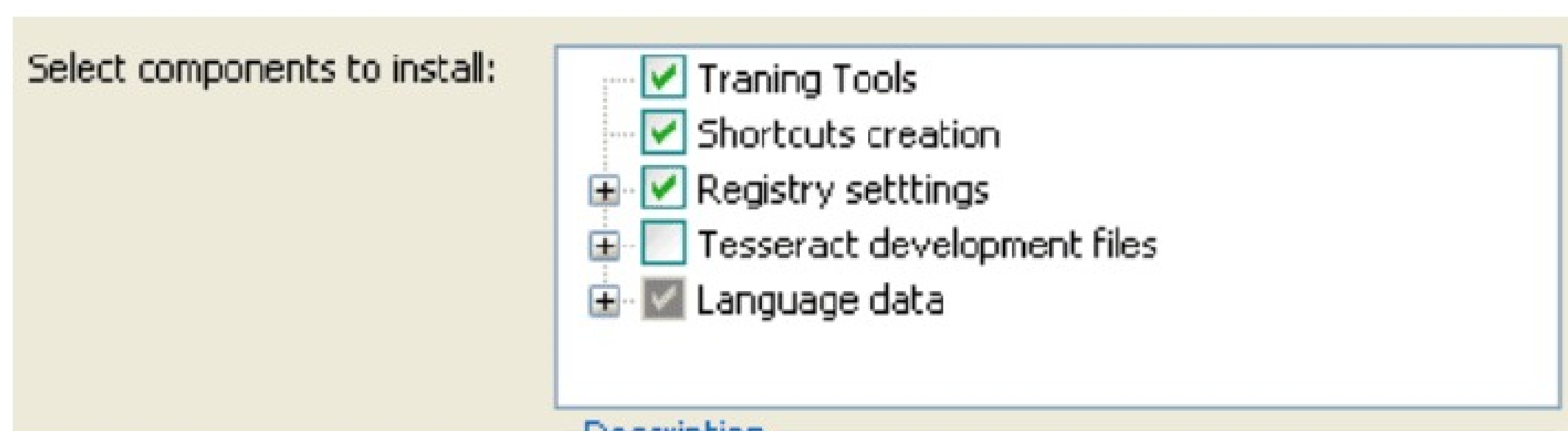


图 12-1　OCR 安装选项

选项“Language data”默认勾选英文，这是识别文字选项。如果要识别其他国家的语言，可自行勾选，但勾选其他语言，在下一步安装时需要访问谷歌下载文件。除此之外，可自行下载 chi_sim.traineddata 文件（中文简体语言包），然后放到 C:\Program Files (x86)\Tesseract-OCR\tessdata 文件夹下即可（上述路径是 OCR 引擎默认的安装路径）。

完成上述安装后，就能在 Python 中使用 pyocr 实现 OCR 识别了，方法如下：

（1）创建 OCR 文件夹，在该文件夹下创建 ocr.py 文件和图片 pic.png，如图 12-2 所示。

（2）打开 ocr.py 文件，输入代码：

```
from PIL import Image
from pyocr import tesseract
# 使用 PIL 打开图片
im = Image.open('pic.png')
# OCR 识别
code = tesseract.image to string(im)
print(code)
```

（3）运行 ocr.py，运行结果如图 12-3 所示。

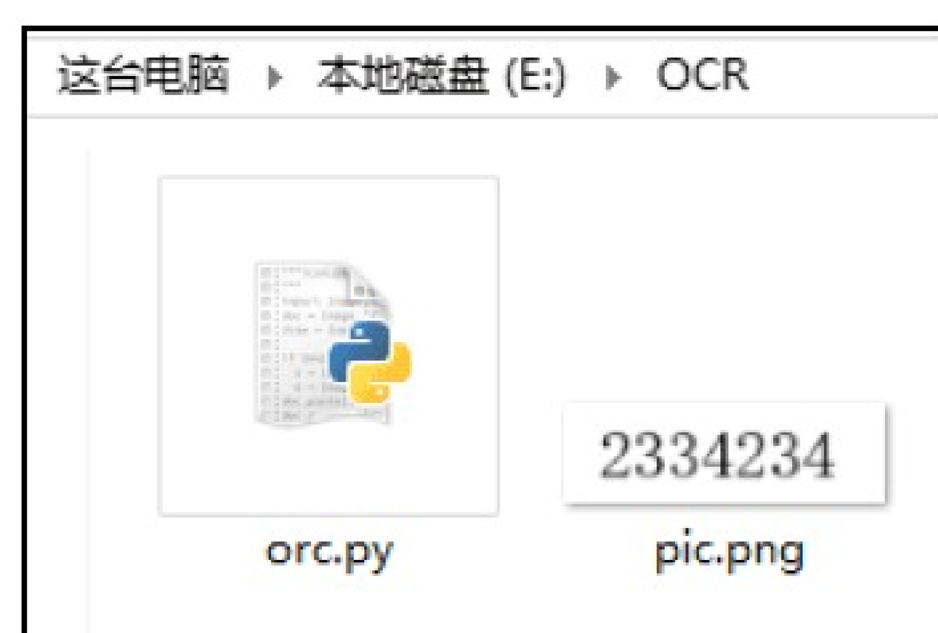


图 12-2　OCR 使用

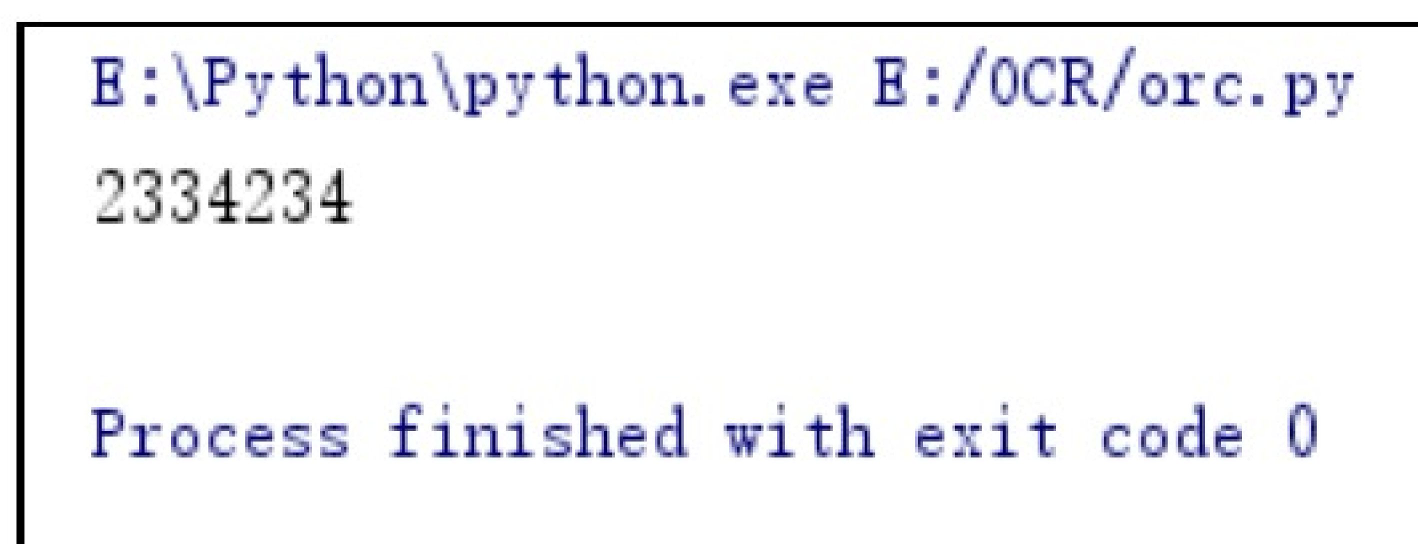


图 12-3　验证码识别结果

在实际使用时，验证码图片不会是一张白底黑字的图片，往往会掺入很多干扰因素，这样会导致识别出来的结果与实际相差甚大。为了提高准确率，可以使用 PIL 模块对图片进行简单的处理，如图 12-4 所示。



红色背景　　　蓝色背景

图 12-4　验证码图片

图 12-4 分别是带红色和蓝色背景的验证码，而且背景颜色带有其他杂色，如果使用上述代码对图片进行识别，识别结果可能与实际完全不相符。此时，可以对图片进行简单的处理，去掉干扰因素，提高识别准确率。图片处理主要由 PIL 模块实现，将图 12-4 中的验证码分别命名为 pic1.png 和 pic2.png 文件，实现代码如下：

```
from PIL import Image
from pyocr import tesseract

pic_list = ['pic1.png','pic2.png']
for i in pic_list:
    im = Image.open(i)
    im = im.convert('L')# 图片转换为灰色图像
    # 保存转换后的图片
    im.save("temp.png")
    code = tesseract.image_to_string(im)
    print(code)
```

PIL 模块打开图片并生成图片对象 im，然后转换图片颜色模式，将带有颜色的图片转换成灰度模式，形成黑→灰→白的过渡，如同黑白照片，最后交给 OCR 引擎识别并返回识别结果。

提　示
颜色模式是将某种颜色表现为数字形式的模型，或者是一种记录图像颜色的方式，分为 RGB 模式、CMYK 模式、HSB 模式、Lab 颜色模式、位图模式、灰度模式、索引颜色模式、双色调模式和多通道模式。

程序运行结果如图 12-5 所示。

```
E:\Python\python.exe E:/OCR/orc.py
IMDS
SXRB

Process finished with exit code 0
```

图 12-5　验证码识别结果

本书只介绍简单的图片处理，不同的图片有不同的处理方法，其目的都为提高 OCR 识别的准确率。除此之外，提高 OCR 准确率还可以对 OCR 引擎进行训练和学习，但两者已经属于人工智能的领域，其涉及较多的知识点，所以就不一一讲解了。

12.3　第三方平台

除了使用 OCR 识别验证码之外，还可以利用第三方平台实现验证码的识别。到目前为止，这是解决验证码最快、最简单的途径，而且有完善的 API 接口，能帮助开发者完成快速开发的需求，

但每次调取 API 接口需要收取少量费用。

验证码识别平台主要有以下两种。

- 打码平台：主要由在线人员识别验证码。开发者只需调用平台 API 接口，一般在 10 秒内返回结果。识别错误或者无法识别不收费。
- AI 开发者平台：主要由人工智能系统识别，准确率取决于系统的智能程度。调用 API 接口每天有免费使用次数，也可以付费使用。目前，主流平台有百度 AI 和腾讯 AI。

本书以打码平台为例，在浏览器上访问 http://www.yundama.com/，注册账号后充值就能调用 API 接口识别验证码，在平台上提供开发文档，代码如下：

```
import json
import time
import requests
class YDMHttp:
    apiurl = 'http://api.yundama.com/api.php'
    username = ''
    password = ''
    appid = '4055'
    appkey = 'c5e26d1a207df586d7aaec21522dd446'
    def   init  (self, name, passwd, app id, app key):
        self.username = name
        self.password = passwd
        self.appid = str(app id)
        self.appkey = app key

    def request(self, fields, files=[]):
        response = self.post url(self.apiurl, fields, files)
        response = json.loads(response)
        return response

    def balance(self):
        data = {
            'method': 'balance',
            'username': self.username,
            'password': self.password,
            'appid': self.appid,
            'appkey': self.appkey
        }
        response = self.request(data)
        if response:
            if response['ret'] and response['ret'] < 0:
                return response['ret']
            else:
                return response['balance']
        else:
            return -9001

    def login(self):
        data = {'method': 'login', 'username': self.username,
               'password': self.password, 'appid': self.appid,
               'appkey': self.appkey}
        response = self.request(data)
```

```
        if response:
            if response['ret'] and response['ret'] < 0:
                return response['ret']
            else:
                return response['uid']
        else:
            return -9001

    def upload(self, filename, codetype, timeout):
        data = {'method': 'upload', 'username': self.username,
                'password': self.password, 'appid': self.appid,
               'appkey': self.appkey, 'codetype': str(codetype),
                'timeout': str(timeout)}
        file = {'file': filename}
        response = self.request(data, file)
        if response:
            if response['ret'] and response['ret'] < 0:
                return response['ret']
            else:
                return response['cid']
        else:
            return -9001

    def result(self, cid):
        data = {'method': 'result', 'username': self.username,
                'password': self.password, 'appid': self.appid,
               'appkey': self.appkey, 'cid': str(cid)}
        response = self.request(data)
        return response and response['text'] or ''

    def decode(self, file_name, code_type, time_out):
        cid = self.upload(file_name, code_type, time_out)
        if cid > 0:
            for i in range(0, time_out):
                result = self.result(cid)
                if result != '':
                    return cid, result
                else:
                    time.sleep(1)
            return -3003, ''
        else:
            return cid, ''

    def post_url(self, url, fields, files=[]):
        for key in files:
            files[key] = open(files[key], 'rb')
        res = requests.post(url, files=files, data=fields)
        return res.text

def code_verificate(name, passwd, file_name,
                     app_id=4055, code_type=1005, time_out=60):
    # name: 云打码注册用户名; passwd: 用户密码; file_name: 需要识别的图片名
    app_key='c5e26d1a207df586d7aaec21522dd446'
    yundama_obj = YDMHttp(name, passwd, app_id, app_key)
    cur_uid = yundama_obj.login()
```

```
        print('uid: %s' % cur_uid)
        rest = yundama_obj.balance()
        print('balance: %s' % rest)
        # 开始识别图片路径、验证码类型 ID、超时时间（秒），并显示识别结果
        cid, result = yundama_obj.decode(file_name, code_type, time_out)
        print('cid: %s, result: %s' % (cid, result))
        return result
    if __name__ == '__main__':
        # 云打码注册的登录用户名（通过用户注册）
        username = 'xxx'
        # 登录密码
        password = 'xxx'
        rs = code_verificate(username, password, 'pincode.png')
```

使用方法：只需在爬虫代码中引用上述文档，然后调用 code_verificate()方法函数，传入已注册的用户信息和需要识别的图片即可。

12.4 本 章 小 结

本章中读者应重点掌握以下内容。

1. 验证码

验证码的作用是加强用户安全性和提高反爬虫机制，有效防止这种问题对某一个特定注册用户用特定程序暴力破解的方式不断地进行登录尝试。

读者要了解解决验证码的以下几种方案：

（1）人工识别验证码，将验证码图片下载到本地，然后靠使用者自行识别并输入识别内容，程序获取输入的内容后，用户完成登录。其特点是开发简单，适合初学者，但过分依赖人为控制，难以实现批量爬取。

（2）通过 Python 调用 OCR 引擎识别验证码。这是最理想的解决方案，但 OCR 准确率较低，需要机器学习不断提高 OCR 的准确率，开发成本相对较高。

（3）调用 API 使用第三方平台识别验证码。开发成本较低，有完善的 API 接口，直接调用即可，识别准确率高，但每次识别需收取小额费用。

2. OCR

OCR 是指使用电子设备（例如扫描仪或数码相机）检查纸上打印的字符，通过检测暗、亮的模式确定其形状，然后用字符识别方法将形状翻译成计算机文字的过程；即针对印刷体字符，采用光学的方式将纸质文档中的文字转换成为黑白点阵的图像文件，并通过识别软件将图像中的文字转换成文本格式，供文字处理软件进一步编辑加工的技术。

Python 中支持的 ORC 模块有 pytesser3 和 pyocr，其原理主要是通过模块功能调用 OCR 引擎识别图片，OCR 引擎再将识别的结果返回到程序中。

3. 验证码识别平台

验证码识别平台主要有以下两种。

（1）打码平台：主要由在线人员识别验证码。开发者只需调用平台 API 接口，一般在 10 秒内返回结果。识别错误或者无法识别不收费。

（2）AI 开发者平台：主要由人工智能系统识别，准确率取决于系统的智能程度。调用 API 接口使用，每天有免费使用次数，也可付费使用，目前主流平台有百度 AI 和腾讯 AI。

第 13 章

数据清洗

13.1 字符串操作

从网页上采集数据后，数据大多数是杂乱无章的，这时需要对采集的数据加工清洗，去掉数据中的一些垃圾数据才能得到我们所需的数据。清洗数据有三种常用的方法：字符串操作、正则表达式和第三方模块库。三种方法在不同场景下有不同优势，取长补短，应根据实际情况选择合理的清洗方法，三种方法同时出现在一个项目也是常见的事情。

下面分别介绍用于清洗数据的字符串操作：截取、替换、查找和分割。

13.1.1 截取

格式：字符串[开始位置:结束位置:间隔位置]。

开始位置是 0，正数代表从左边位置开始，负数代表从右边位置开始，默认代表从 0 开始。结束位置是被截取的字符串位置，空值默认取到字符串尾部。间隔位置默认为 1，截取的内容不做处理；如果设置为 2，就将截取的内容再隔一取数。示例如下：

```
# 字符串截取
str = 'ABCDEFG'
# 截取第一位到第三位的字符
print('截取第一位到第三位的字符: ' + str[0:3:])
# 截取字符串的全部字符
print('截取字符串的全部字符: ' + str[::])
# 截取第七个字符到结尾
print('截取第七个字符到结尾: ' + str[6::])
# 截取从头开始到倒数第三个字符之前
print('截取从头开始到倒数第三个字符之前: ' + str[:-3:])
```

```
# 截取第三个字符
print('截取第三个字符: ' + str[2])
# 截取倒数第一个字符
print('截取倒数第一个字符: ' + str[-1])
# 与原字符串顺序相反的字符串
print('与原字符串顺序相反的字符串: ' + str[::-1])
# 截取倒数第三位与倒数第一位之前的字符
print('截取倒数第三位与倒数第一位之前的字符: ' + str[-3:-1:])
# 截取倒数第三位到结尾
print('截取倒数第三位到结尾: ' + str[-3::])
# 逆序截取
print('逆序截取: ' + str[:-5:-3])
```

13.1.2　替换

格式：字符串.replace('被替换内容', '替换后内容')

要注意的是，使用 replace 替换字符串后仅为临时变量，需重新赋值才能保存。示例如下：

```
str = 'ABCABCABC'
# 单个内容替换
print(str.replace('C', 'V'))
# 输出内容：ABVABVABV

# 字符串替换
print(str.replace('BC', 'WV'))
# 输出内容：AWVAWVAWV
# 替换成特殊符号(空格)
print(str.replace('BC', ' '))
# 输出内容：A A A
```

13.1.3　查找

格式：字符串.find('要查找的内容' [, 开始位置,结束位置])

开始位置和结束位置表示要查找的范围，若为空值，则表示查找所有。找到目标后会返回目标第一位内容所在的位置，位置从 0 开始算，如果没找到，就返回-1。示例如下：

```
str = 'ABCDABC'
# 查找全部
print(str.find('A'))
# 输出内容：0

# 从字符串第 4 个开始查找
print(str.find('A', 3))
# 输出内容：4

# 从字符串第 2 个到第 6 个开始查找，即从'BCDAB'中查找'C'
print(str.find('C', 1, 5))
# 输出内容：2
```

```
# 查找不存在的内容
print(str.find('E'))
# 输出内容：-1
```

除了使用 find 函数查找字符串中某个内容之外，index 函数也能实现同样的功能。index 是在字符串里查找子串第一次出现的位置，类似于字符串的 find 方法，如果查找不到子串，就会抛出异常，而不是返回-1。示例如下：

```
str = 'ABCDABC'
# 查找全部
print(str.index('A'))
# 输出内容：0

# 从字符串第 4 个开始查找
print(str.index('A', 3))
# 输出内容：4

# 从字符串第 2 个到第 6 个开始查找
print(str.index('C', 1, 5))
# 输出内容：2

# 查找不存在的内容
print(str.index('E'))
# 输出内容：ValueError: substring not found
```

13.1.4 分割

格式：字符串.split('分割符',分割次数)

如果存在分割次数，就仅分割成“分割次数+1”个子字符串；如果为空，就默认全部分割。分割后，返回一个列表类型数据。例子如下：

```
str = 'ABCDABC'
# 分割全部
print(str.split('B'))
# 输出内容：['A', 'CDA', 'C']
# 分割一次
print(str.split('B', 1))
# 输出内容：['A', 'CDABC']
```

字符串操作是数据清洗的基本，可以解析 HTML，但纯字符串解析 HTML 会导致代码冗余，不便维护，一般不建议这样操作。字符串操作主要用于个别数据清洗，且数据具有一定的特性。如图 13-1 所示是一个示例。



图 13-1　12306 各个城市的站点信息

在浏览器中输入“https://kyfw.12306.cn/otn/leftTicket/init”，在开发者工具→Network→JS 标签中可找到图 8-1 中的站点信息。根据内容分析，每个城市有 5 个信息，从带有“@”的特殊字符开始，每个信息之间用“|”隔开。如果想要获取第二个和第三个信息，可以根据其特性以“|”进行字符串分割，代码如下：

```
import requests
def city_name():
    # 构建请求头
    headers = {'User-Agent':
               'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 '
               '(KHTML, like Gecko) Chrome/63.0.3218.0 Safari/537.36',
               'Referer':
               'https://kyfw.12306.cn/otn/login/init'}
    url = 'https://kyfw.12306.cn/otn/resources/js/framework/
           station_name.js?station_version=1.9031'
    city_code = requests.get(url, headers=headers, verify=False)
    # 数据使用字符串操作处理
    city_code_list = city_code.text.split("|")
    city_dict = {}
    for k, i in enumerate(city_code_list):
        if '@' in i:
        # 城市名作为字典的键，城市编号作为字典的值
        city_dict[city_code_list[k + 1]]=city_code_list[k + 2].replace(' ', '')
    return (city_dict)
# 输出处理后的数据
print(city_name())
```

除了使用 split 对字符串进行分割之外，在数据赋值之前还要使用 replace()对数据进行替换，主要清洗数据中含有空白的内容。在一些设计不规范的网站中，其 HTML 中的数据经常带有空白内容和一些特殊符号，可以使用 replace()对这类数据进行清洗。

13.2　正则表达式

正则表达式是用于处理字符串的强大工具，拥有自己独特的语法及一个独立的处理引擎，效率上可能不如字符串处理方法，但功能十分强大。得益于这一点，在提供了正则表达式的语言里，正则表达式的语法都是一样的，区别只在于不同的编程语言实现支持的语法数量不同，但不用担心，

不被支持的语法通常是不常用的部分。

学习正则表达式要从两方面着手：正则语法和正则处理函数。

13.2.1 正则语法

正则语法也称元字符，符合正则规则，通常表示一些不寻常的匹配操作，或者通过重复、修改匹配意义来影响正则模式的其他部分。常用语法如表 13-1 所示。

表 13-1 正则表达式元字符和语法

元字符	说明	实例
.	匹配任意字符（不包括换行符）	'abc'>>>'a.c'>>>结果为：'abc'
^	匹配开始位置，多行模式下匹配每一行的开始	'abc'>>>'^abc'>>>结果为：'abc'
$	匹配结束位置，多行模式下匹配每一行的结束	'abc'>>>'abc$'>>>结果为：'abc'
*，+，？	匹配前一个元字符 0 到多次	'abcccd'>>>'abc*'>>>结果为：'abccc'
+	匹配前一个元字符 1 到多次	'abcccd'>>>'abc+'>>>结果为：'abccc'
?	匹配前一个元字符 0 到 1 次	'abcccd'>>>'abc?'>>>结果为：'abc'
{m}	匹配前一个字符 m 次	'abcccd'>>>'abc{3}d'>>>结果为：'abcccd'
{m,n}	匹配前一个字符 m 到 n 次	'abcccd'>>>'abc{2,3}d'>>>结果为：'abcccd'
{m,n}?	匹配前一个字符 m 到 n 次，并且取尽可能少的情况	'abccc'>>>'abc{2,3}?'>>>结果为：'abcc'
\\	对特殊字符进行转义，或者指定特殊序列	'a.c'>>>'a\\.c'>>> 结果为：'a.c'
[]	字符集，一个字符的集合，可匹配其中任意一个字符	'abcd'>>>'a[bc]'>>>结果为：'ab'
|	逻辑表达式“或”，比如 a|b 代表可匹配 a 或者 b	'abcd'>>>'abc|acd'>>>结果为：'abc'
(…)	被括起来的表达式作为一个分组。findall 在有组的情况下只显示组的内容	'a123d'>>>'a(123)d'>>>结果为：'123'
(?#...)	添加注释，括号内为注释内容，特殊构建不作为分组	'abc123'>>>'abc(?#fasd)123'>>>结果为：'abc123'
(?= …)	顺序肯定环视，表示所在位置右侧能够匹配括号内正则	在字符串 'pythonretest' 中 (?=test) 会匹配 'pythonre'
(?!...)	顺序否定环视，表示所在位置右侧不能匹配括号内正则	如果'pythonre'右侧不是字符串' test'，也就是说字符串为 testpythonre，那么(?!test)会匹配' pythonre'
(?<= …)	逆序肯定环视，表示所在位置左侧能够匹配括号内正则	与(?!...)实例一致
(?<!...)	逆序否定环视，表示所在位置左侧不能匹配括号内正则	与(?= …)实例一致

正则表达式特殊序列说明如表 13-2 所示。

表 13-2　正则表达式特殊序列

特殊表达式序列	说明
\A	只在字符串开头进行匹配
\b	匹配位于开头或者结尾的空字符串
\B	匹配不位于开头或者结尾的空字符串
\d	匹配任意十进制数，相当于 [0-9]
\D	匹配任意非数字字符，相当于 [^0-9]
\s	匹配任意空白字符，相当于 [\t\n\r\f\v]
\S	匹配任意非空白字符，相当于 [^ \t\n\r\f\v]
\w	匹配任意数字和字母，相当于 [a-zA-Z0-9_]
\W	匹配任意非数字和字母的字符，相当于 [^a-zA-Z0-9_]
\Z	只在字符串结尾进行匹配

13.2.2　正则处理函数

Python 的正则模块是 re，该模块含有多种正则处理函数，常用的功能函数包括：match、search、findall 和 sub。

1. re.match 函数

re.match 函数尝试从字符串的开头开始匹配一个模式，如果匹配成功，就返回一个匹配成功的对象，否则返回 None。

使用方式：re.match(pattern, string, flags=0)

【参数解释】

- pattern：匹配的正则表达式。
- string：要匹配的字符串。
- flags：标志位，用于控制正则表达式的匹配方式，如是否区分大小写、是否多行匹配等。

参数 flags 的可选值如下。

- re.I(re.IGNORECASE)：忽略大小写。
- re.M(MULTILINE)：多行模式，改变'^'和'$'的行为。
- re.S(DOTALL)：此模式下，'.' 的匹配不受限制，可匹配任何字符，包括换行符。
- re.L(LOCALE)：字符集本地化，为了支持多语言版本的字符集使用环境，比如转义符\w。
- re.U(UNICODE)：使预定字符类 \w \W \b \B \s \S \d \D 取决于 unicode 定义的字符属性。
- re.X(VERBOSE)：详细模式。在这个模式下，正则表达式可以是多行的，忽略空白字符，并可以加入注释。

该函数匹配之后，得出一个 match 对象类型，如果要返回结果，那么可以使用 group()或 groups()匹配对象函数来获取匹配后的结果。示例如下：

```
import re
text = "This is the last one"
res = re.match('(.*) is (.*?) .*', text, re.M | re.I)
if res:
    print("res.group() : ", res.group())
    print("res.group(1) : ", res.group(1))
    print("res.group(2) : ", res.group(2))
    print("res.groups() : ", res.groups())
else:
    print("No match!!")
```

输出结果：

```
res.group() :  This is the last one
res.group(1) :  This
res.group(2) :  the
res.groups() :  ('This', 'the')
```

对于代码中的“(.*)”，“.”代表匹配任意字符，“*”代表匹配前一个字符 0 次或多次，两者结合匹配出“This”；在“is”后面的“(.*?) .*”代表获取 is 后面的全部数据，其中小括号匹配结果，“.*?”用于匹配一个单词“the”，而括号外的“.*”用于匹配任意字符，但不返回给匹配结果。如果对这部分较难理解，读者可以自行比较以下匹配结果：

```
res = re.match('(.*) is (.*?) (.*)', text, re.M | re.I)
res = re.match('(.*) is (.*) (.*)', text, re.M | re.I)
res = re.match('(.*) is (.*)', text, re.M | re.I)
res = re.match('(.*) is (.*?)', text, re.M | re.I)
```

2. re.search 函数

re.search 扫描整个字符串并返回第一次成功匹配的对象，如果匹配失败，就返回 None。

使用方式：re.search(pattern, string, flags=0)

【参数解释】

- pattern：匹配的正则表达式。
- string：要匹配的字符串。
- flags：标志位，用于控制正则表达式的匹配方式，如是否区分大小写、是否多行匹配等，flags 可选值与 match 一样。

匹配结果跟 re.match 函数一样，使用 group()和 groups()方法来获取。

将 re.match 的例子改为 re.search，得出的结果是一致的。search 和 match 的使用方法相似，不过两者运行逻辑不同，两者的主要区别：re.match 只匹配字符串的开始，如果字符串开始不符合正则表达式，匹配就会失败，函数返回 None；而 re.search 匹配整个字符串，直到找到一个匹配的字符串，否则也返回 None。

3. re.findall 函数

re.findall 函数用于获取字符串中所有匹配的字符串，并以列表的形式返回。列表中的元素有如

下几种情况：

① 当正则表达式中含有多个圆括号时，返回的列表元素为多个字符串组成的元组，而且元组中的字符串个数与括号对数相同，并且字符串排放顺序跟括号出现的顺序一致，字符串的内容与每个括号内的正则表达式相对应。

② 当正则表达式中只带有一个圆括号时，返回的列表元素为字符串，并且该字符串的内容与括号中的正则表达式相对应。（注意：返回的列表（字符串）只是圆括号中的内容，不是整个正则表达式所匹配的内容。）

③ 当正则表达式中没有圆括号时，返回的列表中的字符串表示整个正则表达式匹配的内容。

使用方式：re.findall(pattern, string, flags=0)

【参数解释】

- pattern：匹配的正则表达式。
- string：要匹配的字符串。
- flags：标志位，用于控制正则表达式的匹配方式，如是否区分大小写、是否多行匹配等，flags 可选值与 match 一样。

匹配结果不需要使用 group()和 groups()方法来获取。示例如下：

```
import re
# 匹配字符串中所有含有'oo'字符的单词
# 当正则表达式中没有圆括号时，列表中的字符串表示整个正则表达式匹配的内容
find_value = re.findall('\w*oo\w*', 'woo this foo is too')
print(find_value)

# 获取字符串中所有的数字字符串
# 当正则表达式中只带有一个圆括号时，列表中的元素为字符串，
# 并且该字符串的内容与括号中的正则表达式相对应
find_value = re.findall('.*?(\d+).*?', 'adsd12343.jl34d5645fd789')
print(find_value)

# 提取字符串中所有有效的域名地址
# 正则表达式中有多个圆括号时，返回匹配成功的列表中的每一个元素都是由一次匹配
# 成功后，正则表达式中所有括号中匹配的内容组成的元组
add = 'https://www.net.com.edu//action=?asdfsd and other
        https://www.baidu.com//a=b'
find_value = re.findall('((w{3}\.)(\w+\.)+(com|edu|cn|net))', add)
print(find_value)
```

输出结果：

```
['woo', 'foo', 'too']
['12343', '34', '5645', '789']
[('www.net.com.edu', 'www.', 'com.', 'edu'), ('www.baidu.com', 'www.',
'baidu.', 'com')]
```

4. re.sub 函数

re.sub 函数用于替换字符串中的匹配项，如果没有匹配的项，则返回没有匹配的字符串。

使用方式：re.sub(pattern, repl, string, count=0, flags=0)

【参数解释】

- pattern：匹配的正则表达式。
- repl：用于替换的字符串。
- string：要被替换的字符串。
- count：替换的次数。
- flags：标志位，用于控制正则表达式的匹配方式，如是否区分大小写、是否多行匹配等，flags 可选值与 match 一样。

匹配结果不需要使用 group()和 groups()方法来获取。示例如下：

```
import re
# 将手机号的后 4 位替换成 0
replace_value = re.sub('\d{4}$', '0000', '13435423143')
print(replace_value)
# 将代码后面的注释信息去掉
replace_value = re.sub('#.*$', '', 'num = 0 #a number')
print(replace_value)
```

输出结果：

```
13435420000
num = 0
```

上述是正则处理函数的常用函数，除此之外，正则处理函数还有：

- re.split(pattern, string[, maxsplit])：用匹配 pattern 的子串来分割字符串。
- re.subn(pattern, repl, string[, count])：与 sub()函数一样，只是返回结果是一个元组。
- re.escape(string)：把字符串里除了字母和数字以外的字符都加上反斜杆。
- re.finditer(pattern, string[, flags])：搜索字符串，按顺序返回每一个匹配结果的迭代器。

在学习正则表达式时，难点是如何根据字符串内容编写正确的正则表达式，元字符之间不同的组合会产生不同的结果，要熟练掌握正则表达式，必须熟知每个元字符的作用以及正则处理函数的使用方法。

13.3 BeautifulSoup 数据清洗

13.3.1 BeautifulSoup 介绍与安装

BeautifulSoup 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库。其功能简单而强大，容错能力高，文档相对完善，清晰易懂，具有三个特性：

（1）BeautifulSoup 提供了一些简单的方法和 Python 术语，用于检索和修改语法树：一个用于解析文档并提取相关信息的工具包。

（2）BeautifulSoup 自动将输入文档转换为 Unicode 编码，并将输出文档转化为 UTF-8 编码。不需要考虑编码，除非输入文档没有指出其编码并且 BeautifulSoup 无法自动检测到，这时需要指

出原来的编码方式。

（3）BeautifulSoup 位于一些流行的 Python 解析器中，比如 lxml 和 html5lib 的上层，这允许使用不同的解析策略或者牺牲速度来换取灵活性。

BeautifulSoup 的安装涉及第三方的扩展，建议使用 pip 安装。

```
pip install beautifulsoup4
```

Beautiful Soup 支持 Python 标准库中的 HTML 解析器，还支持一些第三方的解析器，常用的解析器有 lxml 和 html5lib。lxml 的安装如下：

```
pip install lxml
```

lxml 是一个用来处理 XML 的第三方 Python 库，它的底层封装了由 C 语言编写的 libxml2 和 libxslt，并以简单、强大的 Python API 兼容并加强了著名的 ElementTree API。

另一个可供选择的解析器是纯 Python 实现的 html5lib，这是一个 Ruby 和 Python 用来解析 HTML 文档的类库，支持 HTML5 以及最大程度兼容桌面浏览器。使用 pip 安装 html5lib：

```
pip install html5lib
```

完成上述安装后，在 CMD（终端）下验证安装是否成功，打开 Python 交互式命令行（输入“Python”，按回车键即可），输入代码验证即可：

```
>>> import html5lib
>>> html5lib.  version
'0.999999999'
>>> import lxml
>>> import bs4
>>> bs4.  version
'4.6.0'
```

每种解析器都有自己的优缺点，对比如表 13-3 所示。

表 13-3　各种解析器的比较

解析器	使用方法	优势	劣势
标准库	BeautifulSoup(html, "html.parser")	内置标准库，速度适中，文档容错能力强	Python3.2 版本前的文档容错能力差
lxml HTML	BeautifulSoup(html, "lxml")	速度快，文档容错能力强	安装 C 语言库
lxml XML	BeautifulSoup(html, "xml")	速度快，唯一支持 XML	安装 C 语言库
html5lib	BeautifulSoup(html, "html5lib")	容错性最强，可生成 HTML5	运行慢，扩展差

在选择解析器的时候，要从实际出发，解析同一个 HTML，不同的解析器因为容错性不同会导致不同的结果，若得到的数据与实际存在差异，出现数据丢失和解析出来的数据与实际不相符，则有可能是解析器在解析 HTML 时出现错误，可选择不同的解析器对错误逐一排查。

13.3.2 BeautifulSoup 的使用示例

下面通过例子说明如何使用 BeautifulSoup，以 MySoup.html 文件内容为例：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title> Python</title>
</head>
<body>
<p id="python">
<a href="/index.html"> Python </a>BeautifulSoup 的使用
</p>
<p class="myclass">
<a href="http://www.baidu.com/">这是</a> 一个指向百度的页面的 URL。
</p>
</body>
</html>
```

在文件 MySoup.html 的同一目录创建 Soup_py.py 文件，Soup_py.py 的代码如下：

```
# 引入 BeautifulSoup
from bs4 import BeautifulSoup
# 读取 MySoup.html 文件
Open file = open('MySoup.html', 'r', encoding='utf-8')
# 将 MySoup.html 的内容赋值给 Html_Content，并关闭文件
Html Content = Open file.read()
Open file.close()
# 使用 html5lib 解释器解释 Html_Content 的内容
soup = BeautifulSoup(Html Content, "html5lib")
# 输出 title
print('html title is ' + soup.title.getText())
# 查找第一个标签 p,并输出
find p = soup.find('p', id="python")
print('the first <p> is ' + find p.getText())
# 查找全部标签 p,并输出
find all p = soup.find all('p')
for i, k in enumerate(find all p):
    print('the ' + str(i + 1) + ' p is ' + k.getText())
```

运行 Soup_py.py，结果如图 13-2 所示。

```
html title is  Python网络爬虫实战揭秘
the first <p> is
 Python BeautifulSoup的使用

the 1 p is
 Python BeautifulSoup的使用

the 2 p is
这是 一个指向百度的页面的链接。
```

图 13-2　BeautifulSoup 解析 HTML

代码运行时，先读取 HTML 文件的内容，将内容定义到一个 BeautifulSoup 对象中，并使用 html5lib 解析 HTML 内容，然后使用 Beautiful Soup 内置的方法找出标题的值和<p>的值。

前面的例子只是简单介绍了 Beautiful Soup 的基本用法，下面以 Python 的交互式命令行演示 Beautiful Soup 的更多用法（在 CMD（终端）中输入“Python”，按回车键可进入 Python 的交互式命令行）。

（1）查找全部标签，代码如下：

```
Html content = """<html><head><title> Python</title></head>
<p class="title"><b>Beautiful Soup 的学习</b></p>
<p class=” study” >学习网址：http://blog.csdn.net/huangzhang 123
<a href="www.xxx.com" class="abc" id="try1">web 开发</a>,
<a href=" www.ccc.com " class="bcd" id="try2">网络爬虫</a> and
<a href=" www.aaa.com " class="efg" id="try3">人工智能</a>;
</p>
<p class="other">...</p>"""

from bs4 import BeautifulSoup
soup = BeautifulSoup(Html content, "html5lib")
# 以下是查找某标签的方法：
# 获取头部的信息，返回<head></head>之间的全部内容
soup.head
# 获取 title 的信息，返回<title></title>之间的全部内容
soup.title
# 这是一个获取 tag 的小窍门，可以在文档树的 tag 中多次调用这个方法。
# 下面的代码可以获取<body>标签中的第一个<b>标签
# 也就是说，soup 不一定是整个 html 的内容，可以先定位某部分，然后用这个简洁的方式获取
# 返回"<b>Beautiful Soup 的学习</b>"
soup.body.b
# 直接指定标签类别，返回第一个标签的内容。返回 "<a href="www.xxx.com" class="abc"
# id = "try1">web 开发</a>"
soup.a
# 获取第一个标签 a
soup.find_all('a')
#[<a href="www.xxx.com" class="abc" id="try1">web 开发</a>,
#<a href=" www.ccc.com " class="bcd" id="try2">网络爬虫</a>,
#<a href=" www.aaa.com " class="efg" id="try3">人工智能</a>]
```

变量 Html_content 是一个 HTML 内容，其格式为字符串，Beautiful Soup 对 Html_content 生成对象 soup。数据获取是从 soup 对象获取，比如获取 head 和 title，可从 soup.head 和 soup.title 直接获取。想获取某个标签值，如 soup.a，返回的数据格式是<class 'bs4.element.Tag'>，这是 Beautiful Soup 的格式，代表第一个标签的全部内容，若想获取其标签在网页上显示的内容（去除 HTML 代码），则可通过以下方法：

① 通过 getText()获取标签的值。例如，soup.a.getText() 返回的是“web 开发”。

② 通过 str()方式转换为字符串。例如，str(soup.a) 返回的是“web 开发”，然后使用字符串截取获取的数据。

（2）获取某标签的属性值，在上述例子中，soup.a 可以获取第一个 HTML 的标签 a，如果想获取该标签里面的属性值，沿用上述变量 Html_content，实现代码如下：

```
    soup = BeautifulSoup(Html content, "html5lib")
    print(soup.a['class'])
    # 输出内容：'abc'
```

值得注意的是，在 HTML 中，class 属性可带有多个 CSS 样式，如果 HTML 的属性含有多个 CSS 样式，BeautifulSoup 会以列表的格式返回结果。例子如下：

```
    soup = BeautifulSoup('<a href="www.xxx.com" class="abc bcd">web 开发</a>',
"html5lib")
    print(soup.a['class'])
    # 输出内容：[ ‘abc’ , ‘bcd’ ]
```

（3）精准查找

上述例子只能返回第一个标签 a，如果想获取第 N 个标签 a 或者精确定位到某个标签，就只能使用其他方法实现。沿用上述变量 Html_content，实现精确定位标签 a，方法如下：

```
    soup.find_all('a', id=" try3")
    soup.find_all('a', class ="efg", id=" try3")
    soup.find_all('a', href == re.compile("aaa"))
```

以上三种方式都可以定位到人工智能这个标签。

① 第一种是通过一个属性定位，只要是标签里具有的属性都可以定位到。

② 第二种在第一种的基础上增加了一种属性，也就是多个属性一起定位，这样更加精准。

③ 第三种是通过正则表达式进行模糊匹配，这个适合属性多变时使用。

在 BeautifulSoup 中，find()和 find_all()的使用方法一样。两者的区别在于：

① find_all()返回的结果是包含一个或多个元素的列表；而 find()方法返回的是第一个符合要求的结果，格式为字符串。

② 若 find_all()没有找到目标，则返回空列表；而 find()方法找不到目标时，返回 None。

（4）Beautiful Soup 支持大部分的 CSS 选择器。

CSS 样式定义由两部分组成，形式为：[code] 选择器{样式} [/code]。

在{}之前的部分就是“选择器”。“选择器”指明了{}中“样式”的作用对象，也就是“样式”作用于网页中的哪些元素。

CSS 选择器主要是由前端的 CSS 编写的，这里简单介绍一下 BeautifulSoup 的 CSS 选择器的用法。

- 通过 id 查找：soup.select("#try3")。
- 通过 class 查找：soup.select(".efg")。
- 通过属性查找：soup.select('a[class="efg"]')。

上述三个方法也可以返回人工智能这个标签，与 find_all 实现的功能一样。

BeautifulSoup 在爬虫开发中担任着数据清洗的角色，掌握上述使用方法就能解决绝大部分的网站数据清洗问题。

13.4　本 章 小 结

数据清洗是爬虫开发重要的一环，主要衔接了数据爬取和数据入库。常用的数据清洗方法有：字符串操作、正则表达式和第三方模块（库）。

常用数据清洗的字符串操作有截取、替换、查找和分割。

- 截取：字符串[开始位置:结束位置:间隔位置]。
- 替换：字符串.replace('被替换内容', '替换后内容')。
- 查找：字符串.find('要查找的内容' [，开始位置，结束位置])。
- 分割：字符串.split('分割符',分割次数)。

正则表达式包含正则语法和正则处理函数。

- 正则语法：也称元字符，这类符号代表正则规则，通常表示一些不寻常的匹配操作，或者通过重复、修改匹配意义来影响正则模式的其他部分。
- 正则处理函数：Python 的正则模块是 re，该模块含有多种正则处理函数，功能函数包括：
  （1）re.match(pattern, string, flags=0)
  （2）re.search(pattern, string, flags=0)
  （3）re.findall(pattern, string, flags=0)
  （4）re.sub(pattern, repl, string, count=0, flags=0)
  （5）re.split(pattern, string[, maxsplit])
  （6）re.subn(pattern, repl, string[, count])
  （7）re.escape(string)
  （8）re.finditer(pattern, string[, flags])

BeautifulSoup 是一个可以从 HTML 或 XML 文件中提取数据的 Python 库。常用的数据清洗函数如下。

- 查找全部标签：soup.a，返回第一个标签 a。
- 获取定位元素（标签）的值：soup.a.getText()，获取第一个标签 a 的值。
- 获取标签属性：soup.a['href']，获取整个 HTML 第一个标签 a 的 href 属性值。
- 精准查找：find_all()和 find()。
  - 属性定位：soup.find_all('a', id=" try3")。
  - 多属性定位：soup.find_all('a', class_="efg", id=" try3")。
  - 正则表达式模糊匹配：soup.find_all('a', href == re.compile("aaa"))。
- CSS 选择器。
  - 通过 id 查找：soup.select("#try3")。
  - 通过 class 查找：soup.select(".efg")。
  - 通过属性查找：soup.select('a[class="efg"]')。

第 14 章

文档数据存储

14.1　CSV 数据的写入和读取

常用的数据存储介质有文件、关系式数据库和非关系式数据库。文本文档存储适用于具有时效性的数据，如股市行情、商品信息和排行榜信息等，这类数据具有动态变化性质，非特殊要求下，建议存放文件。

Python 标准库自带 CSV 模块，不用自行安装。数据写入 CSV 的代码如下：

```
import csv
# 若存在文件，则打开 csv 文件；若不存在，则新建文件
# 若不设置 newline=''，则每行数据会隔一行空白行
csvfile = open('csv_test.csv', 'w', newline='')
# 将文件加载到 csv 对象中
writer = csv.writer(csvfile)
# 写入一行数据
writer.writerow(['姓名', '年龄', '电话'])
# 多行数据写入
data = [
    ('小 P', '18', '138001380000'),
    ('小 Y', '22', '138001380000')
]
writer.writerows(data)
# 关闭 csv 对象
csvfile.close()
```

写入 CSV 时使用 open 函数打开文件，open 函数最好设置参数“newline”为空，否则每次写入一行数据，数据之间就会空出一行空白行。将打开的文件对象加载到 CSV 对象中，写入数据分为单行写入和多行写入，对应函数分别是 writerow 和 writerows。

读取 CSV 文件，读取函数有 reader 和 DictReader，两者都是接收一个可迭代的对象，返回一个生成器。reader 函数是将一行数据以列表形式返回；DictReader 函数返回的是一个字典，字典的

值是单元格的值，而字典的键则是这个单元格的标题（列头）。代码如下：

```
import csv
csvfile = open('csv test.csv', 'r')
# 以列表形式输出
reader = csv.reader(csvfile)
# 以字典形式输出，第一行作为字典的键
# reader = csv.DictReader(csvfile)
rows = [row for row in reader]
print(rows)
```

上述代码用于获取全部数据，如果要获取某行数据，就可以循环全部数据，再对每行数据做一个判断，判断是否符合筛选条件，代码如下：

```
import csv
csvfile = open('csv test.csv', 'r')
# 以列表形式输出
reader = csv.reader(csvfile)
for row in reader:
    if '小 P' in row:
        print(row)
# 以字典形式输出，第一行作为字典的键
# reader = csv.DictReader(csvfile)
# for row in reader:
#     if row['姓名']== '小 P':
#         print(row)
```

要获取某行数据，使用不同函数会有不同的判断方式，reader 函数返回的是列表，DictReader 返回的是字典，要根据某个值判断筛选，所采用的方法也不一样。CSV 的存储相对较为简单，而且实用性比较强。

14.2　Excel 数据的写入和读取

Python 操作的 Excel 库有 xlrd、xlwt、pyExcelerator 和 openpyxl。其中，pyExcelerator 只支持 2003 版本，openpyxl 只支持 2007 版本，xlrd 支持 Excel 任何版本的读取，xlwt 支持 Excel 任何版本的写入。

为了版本的兼容性，大多数开发人员选择使用 xlrd 和 xlwt 操作 Excel。xlrd 和 xlwt 的安装如下：

```
pip install xlrd
pip install xlwt
```

完成安装后，在 Python 交互式命令行输入验证代码：

```
>>> import xlwt
>>> import xlrd
>>> xlrd.  VERSION
'1.1.0'
>>> xlwt.  VERSION
'1.3.0'
```

Excel 的写入相对比 CVS 复杂，Excel 可以实现设置数据格式、合并单元格、设置公式和插入图片等功能。使用 xlwt 实现上述功能的代码如下：

```
import xlwt
# 新建一个 Excel 文件
wb = xlwt.Workbook()
# 新建一个 Sheet
ws = wb.add sheet('Python', cell overwrite ok=True)
# 定义字体对齐方式对象
alignment = xlwt.Alignment()
# 设置水平方向
# HORZ GENERAL, HORZ LEFT, HORZ CENTER, HORZ RIGHT, HORZ FILLED
# HORZ JUSTIFIED, HORZ CENTER ACROSS SEL, HORZ DISTRIBUTED
alignment.horz = xlwt.Alignment.HORZ CENTER
# 设置垂直方向
# VERT TOP, VERT CENTER, VERT BOTTOM, VERT JUSTIFIED, VERT DISTRIBUTED
alignment.vert = xlwt.Alignment.VERT CENTER
# 定义格式对象
style = xlwt.XFStyle()
style.alignment = alignment
# 合并单元格 write_merge(开始行, 结束行, 开始列, 结束列, 内容, 格式)
ws.write_merge(0, 0, 0, 5, 'Python 网络爬虫', style)

# 写入数据 wb.write(行,列,内容)
for i in range(2, 7):
    for k in range(5):
        ws.write(i, k, i+k)
    # Excel 公式 xlwt.Formula
    ws.write(i, 5, xlwt.Formula('SUM(A'+str(i+1)+':E'+str(i+1)+')'))

# 插入图片，insert bitmap(img, x, y, x1, y1, scale x=0.8, scale y=1)
# 图片格式必须为 bmp
# x 表示行数，y 表示列数
# x1 表示相对原来位置向下偏移的像素
# y1 表示相对原来位置向右偏移的像素
# scale_x、scale_y 缩放比例
ws.insert bitmap('E:\\test.bmp', 9, 1, 2, 2, scale x=0.3, scale y=0.3)

# 保存文件
wb.save('file.xls')
```

代码依次实现的功能如下。

- 设置字体水平垂直居中：该功能实现共分为两步，第一步是定义 xlwt.Alignment()对象，分别设置其水平方向和垂直方向的属性；第二步是定义 xlwt.XFStyle()对象，将设置好的 Alignment()对象赋予 XFStyle()对象。在写入数据的时候，XFStyle()对象作为 write_merge()方法的参数。
- 合并单元格：主要由 write_merge(开始行, 结束行, 开始列, 结束列, 内容, 格式)方法实现。
- 生成表格并计算每行总和：通过嵌套循环生成 5 行 6 列的表格，第 1 到第 5 列的数据写入由 write()方法实现；第 6 列数据是累计求和，由 Excel 自带公式实现。
- 插入图片：图片插入是由 insert_bitmap(img, x, y, x1, y1, scale_x=0.8, scale_y=1)实现的，图

片格式必须为 bmp，否则无法插入并提示错误。

把数据写入 Excel 的整体思路如下：

（1）xlwt 创建生成临时 Excel 对象。
（2）添加 WorkSheets 对象。
（3）单元格的位置由行列索引决定，索引从 0 开始。
（4）数据写入主要由 write_merge()和 write()实现，两者分别是合并单元格再写入和单元格写入。
（5）设置数据格式是在写入（write_merge()和 write()）的数据中传入参数 style。

运行程序，结果如图 14-1 所示。

	A	B	C	D	E	F
1	Python网络爬虫					
2						
3	2	3	4	5	6	20
4	3	4	5	6	7	25
5	4	5	6	7	8	30
6	5	6	7	8	9	35
7	6	7	8	9	10	40

Python

图 14-1　在 Excel 中写入数据

除此之外，xlwt 还可以设置单元格背景颜色、添加单元格边框、设置单元格高宽度、设置字体颜色和数据类型等，由于篇幅较大，本书就不一一讲解了。

接着读取 Excel 数据，由 xlrd 模块实现，我们以上述已生成的 Excel 为读取目标，代码如下：

```
import xlrd
wb = xlrd.open_workbook('file.xls')
# 获取 Sheets 总数
ws_count = wb.nsheets
print('Sheets 总数：', ws_count)
# 通过索引顺序获取 Sheets
# ws = wb.sheets()[0]
# ws = wb.sheet_by_index(0)
# 通过 Sheets 名获取 Sheets
ws = wb.sheet_by_name('Python')
# 获取整行的值（以列表返回内容）
row_value = ws.row_values(3)
print('第 4 行数据：', row_value)
# 获取整列的值（以列表返回内容）
row_col = ws.col_values(3)
print('D 列数据：', row_col)
```

```
# 获得所有行列
nrows = ws.nrows
ncols = ws.ncols
print('总行数：', nrows, '，总列数：', ncols)

# 获取某个单元格内容 cell(行, 列)
cell_F3 = ws.cell(2, 5).value
print('F3 内容：', cell_F3)

# 使用行列索引获取某个单元格内容
row_F3 = ws.row(2)[5].value
col_F3 = ws.col(5)[2].value
print('F3 内容：', row_F3, 'F3 内容：', col_F3)
```

运行程序，结果如图 14-2 所示。

```
Sheets总数： 1
第4行数据： [3.0, 4.0, 5.0, 6.0, 7.0, 25.0]
D列数据： ['', '', 5.0, 6.0, 7.0, 8.0, 9.0]
总行数： 7 ，总列数： 6
F3内容： 20.0
F3内容： 20.0 F3内容： 20.0
```

图 14-2　从 Excel 中读取数据

读取 Excel 的数据思路大致如下：

（1）xlrd 生成 Workbook 对象，并指向 Excel 文件。

（2）选择 Workbook 里某个 WorkSheets 对象。

（3）获取 WorkSheets 里数据已占用的总行数和总列数（某个单元格数据）。

（4）循环总行数和总列数，读取每一个单元格的数据。

14.3　Word 数据的写入和读取

将数据存储在 Word 文档中，一般以文章、新闻报道和小说这类文字内容较长的数据为主。Python 读写 Word 需要第三方库扩展支持，使用 pip 安装：

```
pip install python-docx
```

模块安装后，验证模块是否安装成功，在 Python 交互式命令行输入验证代码：

```
>>> import docx
>>> docx.__version__
'0.8.6'
```

下面通过例子来讲述如何将数据写入 Word 文档，代码如下：

```
# 数据写入
```

```
from docx import Document
from docx.shared import Inches
# 创建对象
document = Document()
# 添加标题，其中“0”代表标题类型，共有 4 种类型，具体可在 Word 的“开始”→“样式”中查看
document.add_heading('Python 爬虫', 0)
# 添加正文内容并设置部分内容格式
p = document.add_paragraph('Python 爬虫开发-')
# 设置内容加粗
p.runs[0].bold = True
# 添加内容并加粗
p.add_run('数据存储-').bold = True
# 添加内容
p.add_run('Word-')
# 添加内容并设置字体斜体
p.add_run('存储实例。').italic = True
# 添加正文，设置“样式”→“明显引用”
document.add_paragraph('样式'-'明显引用', style='IntenseQuote')
# 添加正文，设置“项目符号”
document.add_paragraph(
    '项目符号 1', style='ListBullet'
)
document.add_paragraph(
    '项目符号 2', style='ListNumber'
)
# 添加图片
document.add_picture('test.png', width=Inches(1.25))
# 添加表格
table = document.add_table(rows=1, cols=3)
hdr_cells = table.rows[0].cells
hdr_cells[0].text = 'Qty'
hdr_cells[1].text = 'Id'
hdr_cells[2].text = 'Desc'
for item in range(2):
    row_cells = table.add_row().cells
    row_cells[0].text = 'a'
    row_cells[1].text = 'b'
    row_cells[2].text = 'c'
# 保存文件
document.add_page_break()
document.save('test.docx')
```

在 Word 中写入数据的整体思路如下：

（1）创建生成临时 Word 对象。

（2）分别使用 add_paragraph()和 add_heading()对 Word 对象添加标题和正文内容。

（3）如果想设置正文内容的字体加粗和斜体等，可以将正文内容 p 对象的属性 runs[0].bold 和 add_run('XX').italic 设置为 True。

（4）如果要插入图片和添加表格，可以在 Word 对象中使用方法 add_picture()和 add_table()。

（5）完成数据写入，需要将 Word 对象保存成 Word 文件。

读取 Word 数据比写入数据相对简单，因为不用设置内容格式，直接获取数据即可。实现代码

如下：

```
# 数据读取
import docx
def readDocx(docName):
    fullText = []
    doc = docx.Document(docName)
    # 读取全部内容
    paras = doc.paragraphs
    # 将每行数据存入列表
    for p in paras:
        fullText.append(p.text)
        # 将列表数据转换成字符串
    return '\n'.join(fullText)
print(readDocx('test.docx'))
```

在 Word 中读取数据的整体思路如下：

（1）生成 Word 对象，并指向 Word 文件。

（2）使用 paragraphs()获取 Word 对象全部内容。

（3）循环 paragraphs 对象，获取每行数据并写入列表。

（4）将列表转换为字符串，每个列表元素使用换行符连接，转换后数据的段落布局与 Word 文档相似。

14.4 本章小结

写入和读取 CSV、Excel 和 Word 中的数据是编写爬虫程序的重要内容，存入 CSV、Excel 和 Word 中的数据一般具体动态变化性质，有一定的时效性，适用于股市行情、商品信息、新闻报道和排行榜信息等。本章主要讲解了 CSV、Excel 和 Word 中数据的写入和读取方法，要点如下：

1．CSV 写入数据的整体思路：

（1）open 函数打开 CSV 文件，模式为 w（一般设置 newline=''），生成 file 对象。

（2）CSV 模块的 writer()方法加载对象 file。

（3）使用 writerow()（writerows()）写入一行（多行）数据。

2．CSV 读取数据的整体思路：

（1）open 函数打开 CSV 文件，模式为 r，生成 file 对象。

（2）CSV 模块的 reader()方法加载对象 file。

（3）使用 reader（DictReader）读取数据。

（4）reader 和 DictReader 区别：两者是接收一个可迭代的对象，返回一个生成器，reader 函数是将一行数据以列表形式返回；DictReader 函数返回的是一个字典，字典的值是单元格的值，而字典的键则是这个单元格的标题（即列头）

3．Excel 写入数据的整体思路：

（1）xlwt 创建生成临时 Excel 对象。

（2）添加 WorkSheets 对象。

（3）单元格的位置由行列索引决定，索引从 0 开始。

（4）数据写入主要由 write_merge()和 write()实现，两者分别是合并单元格再写入和单元格写入。

（5）设置数据格式是在写入（write_merge()和 write()）数据传入参数 style。

4．Excel 读取数据的整体思路：

（1）xlrd 生成 Workbook 对象，并指向 Excel 文件。

（2）选择 Workbook 里某个 WorkSheets 对象。

（3）获取 WorkSheets 里数据已占用的总行数和总列数（某个单元格数据）。

（4）循环总行数和总列数，读取每一个单元格的数据。

5．Word 写入数据的整体思路：

（1）创建生成临时 Word 对象并使用以下方法添加内容：

（2）add_heading()添加标题。

（3）add_paragraph()添加正文内容。

（4）add_picture()插入图片。

（5）add_table()添加表格。

6．Word 读取数据的整体思路：

（1）生成 Word 对象，并指向 Word 文件。

（2）paragraphs()获取 Word 对象全部内容。

（3）循环 paragraphs 对象，获取每行数据并写入列表。

（4）将列表转换为字符串，每个列表元素使用换行符连接，转换后数据的段落布局与 Word 文档相似。

第 15 章

ORM 框架

15.1　SQLAlchemy 介绍与安装

15.1.1　操作数据库的方法

开发人员经常接触的关系数据库主要有 MySQL、Oracle、SQL Server、SQLite 和 PostgreSQL，操作数据库的方法大致有以下两种：

（1）直接使用数据库接口连接。在 Python 的关系数据库连接模块中，分别有 pymysql、cx_Oracle、pymssql、sqlite3 和 psycopg2。通常，这类数据库的操作步骤都是连接数据库、执行 SQL 语句、提交事务、关闭数据库连接。每次操作都需要 Open/Close Connection，如此频繁地操作对于整个系统无疑是一种浪费。对于一个企业级的应用来说，这无疑是不科学的开发方式。

（2）通过 ORM（Object/Relation Mapping，对象-关系映射）框架来操作数据库。这是随着面向对象软件开发方法的发展而产生的，面向对象的开发方法是当今企业级应用开发环境中的主流开发方法，关系数据库是企业级应用环境中永久存放数据的主流数据存储系统。对象和关系数据是业务实体的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。内存中的对象之间存在关联和继承关系，而在数据库中，关系数据无法直接表达多对多关联和继承关系。因此，ORM 系统一般以中间件的形式存在，主要实现程序对象到关系数据库数据的映射。

在实际工作中，企业级开发都是使用 ORM 框架来实现数据库持久化操作的，所以作为一个开发人员，很有必要学习 ORM 框架。

15.1.2　SQLAlchemy 框架介绍

常用的 ORM 框架模块有 SQLObject、Stom、Django 的 ORM、peewee 和 SQLAlchemy。本节主要讲述 Python 的 ORM 框架——SQLAlchemy。SQLAlchemy 是 Python 编程语言下的一款开源软

件，提供 SQL 工具包及对象-关系映射工具，使用 MIT 许可证发行。

SQLAlchemy 采用简单的 Python 语言，为高效和高性能的数据库访问设计，实现了完整的企业级持久模型。SQLAlchemy 的理念是，SQL 数据库的量级和性能重要于对象集合，而对象集合的抽象又重要于表和行。因此，SQLAlchmey 采用类似 Java 里 Hibernate 的数据映射模型，而不是其他 ORM 框架采用的 Active Record 模型。不过，Elixir 和 declarative 等可选插件可以让用户使用声明语法。

SQLAlchemy 首次发行于 2006 年 2 月，是 Python 社区中被广泛使用的 ORM 工具之一，不亚于 Django 的 ORM 框架。

SQLAlchemy 在构建于 WSGI 规范的下一代 Python Web 框架中得到了广泛应用，是由 Mike Bayer 及其开发团队开发的一个单独的项目。使用 SQLAlchemy 等独立 ORM 的一个优势就是允许开发人员首先考虑数据模型，并能决定稍后可视化数据的方式（采用命令行工具、Web 框架还是 GUI 框架）。这与先决定使用 Web 框架或 GUI 框架，再决定如何在框架允许的范围内使用数据模型的开发方法极为不同。

SQLAlchemy 的一个目标是提供能兼容众多数据库（如 SQLite、MySQL、Postgres、Oracle、MS-SQL、SQLServer 和 Firebird）的企业级持久性模型。

15.1.3　SQLAlchemy 的安装

安装 SQLAlchemy 时，建议直接使用 pip 安装。

```
pip install SQLAlchemy
```

除了通过 pip 安装外，也可以在 www.lfd.uci.edu/~gohlke/pythonlibs/#sqlalchemy 下载 SQLAlchemy 版本的 whl 文件，whl 文件可使用 pip 安装，在 CMD（终端）中切换到 whl 文件所在路径，输入安装指令：

```
pip install SQLAlchemy-1.2.14-cp37-cp37m-win_amd64.whl
```

使用 SQLAlchemy 连接数据库实质上还是通过数据库接口实现连接，安装 SQLAlchemy 后还需要安装对应数据库的接口模块，下面以 MySQL 为例安装 pymysql 模块：

```
pip install pymysql
```

完成安装后，打开 CMD 窗口，通过导入模块测试是否安装成功：

```
>>> import sqlalchemy
>>> sqlalchemy.  version
'1.1.14'
>>> import pymysql
>>> pymysql.  version
'0.9.2'
```

15.2 连接数据库

在使用 SQLAlchemy 连接数据库之前，先简单介绍一下数据库系统环境，数据库系统版本信息如图 15-1 所示。



图 15-1 MySQL 信息

使用的数据库是本地数据库，端口是默认端口 3306，是通过 MySQL 工作台创建并命名为 test 的数据库，如图 15-2 所示。

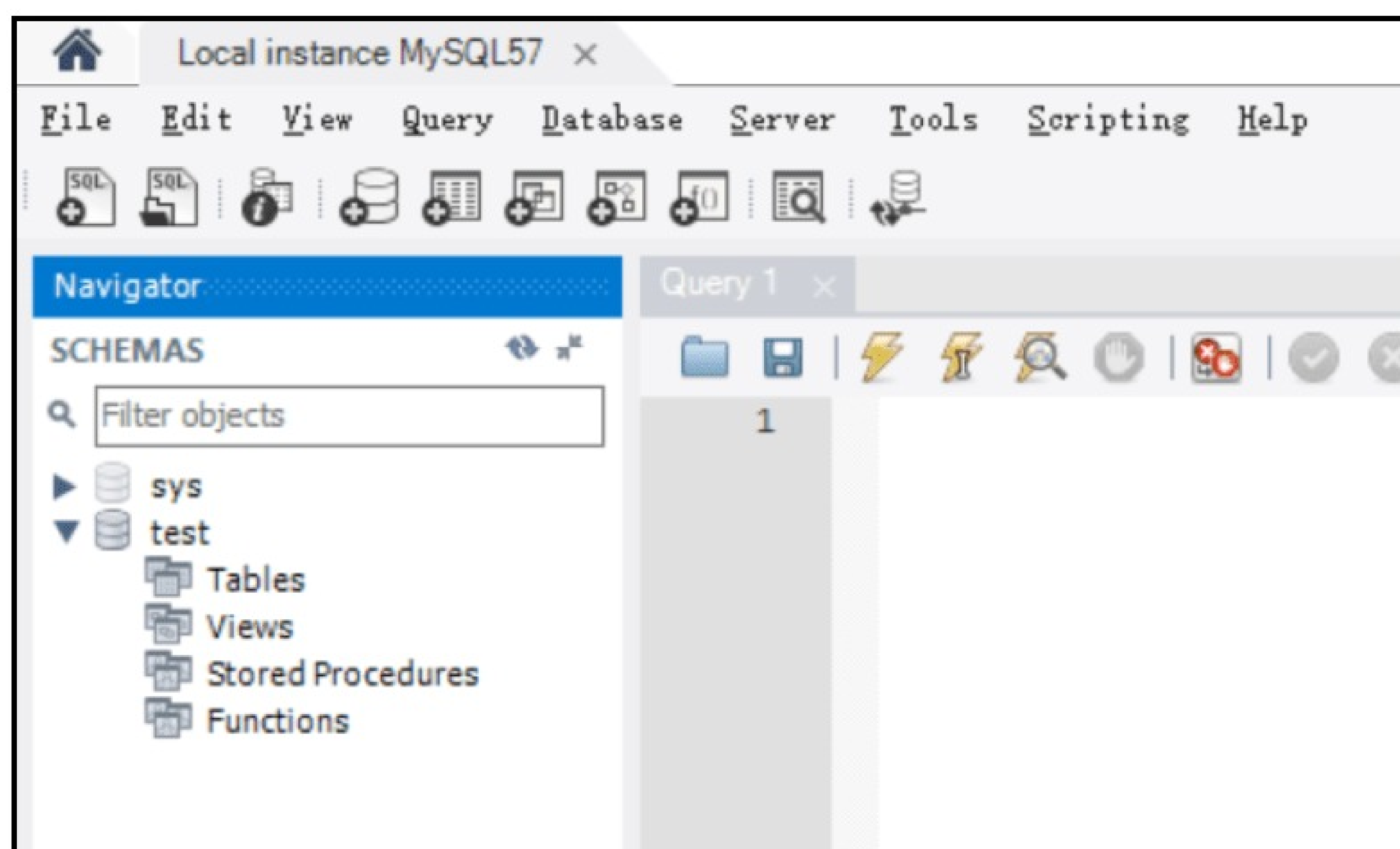


图 15-2 数据库信息

SQLAlchemy 连接数据库使用数据库连接池技术，原理是在系统初始化的时候，将数据库连接作为对象存储在内存中，当用户需要访问数据库时，并非建立一个新的连接，而是从连接池中取出一个已建立的空闲连接对象。使用完毕后，用户也并非将连接关闭，而是将连接放回连接池中，以供下一个请求访问使用。而连接的建立、断开都由连接池自身来管理。同时，还可以通过设置连接

池的参数来控制连接池中的初始连接数、连接的上下限数以及每个连接的最大使用次数、最大空闲时间等。也可以通过其自身的管理机制来监视数据库连接的数量、使用情况等。

通过了解 SQLAlchemy 的原理有利于理解 SQLAlchemy 连接数据库的代码，代码如下：

```
    from sqlalchemy import create_engine
    engine=create_engine("mysql+pymysql://root:110@localhost:3306/test?charset
=utf8",echo=True)
```

导入 SQLAlchemy 的 create_engine 模块，设置数据库指令和参数后可实现连接，上述代码是常用的连接方式。create_engine 的参数设置说明如下。

- mysql+pymysql://root:110@localhost:3306/test：mysql 指明数据库系统类型，pymysql 是连接数据库接口的模块，root 是数据库系统用户名，110 是数据库系统密码，localhost:3306 是本地的数据库系统和数据库端口，test 是数据库名称。
- echo=True：用于显示 SQLAlchemy 在操作数据库时所执行的 SQL 语句情况，相当于一个监视器，可以清楚知道执行情况，如果设置为 False，就可以关闭。
- pool_size：设置连接数，默认设置 5 个连接数，连接数可以根据实际情况进行调整，在一般的爬虫开发中，使用默认值已足够。
- max_overflow：默认连接数为 10。当超出最大连接数后，如果超出的连接数在 max_overflow 设置的访问内，超出的部分还可以继续连接访问，在使用过后，这部分连接不放在 pool（连接池）中，而是被真正关闭。
- pool_recycle：连接重置周期，默认为-1，推荐设置为 7200，即如果连接已空闲 7200 秒，就自动重新获取，以防止 connection 被关闭。
- pool_timeout：连接超时时间，默认为 30 秒，超过时间的连接都会连接失败。
- ?charset=utf8：对数据库进行编码设置，能对数据库进行中文读写，如果不设置，在进行数据添加、修改和更新等时，就会提示编码错误。

完整的连接数据库代码如下：

```
    from sqlalchemy import create_engine
    engine=create_engine("mysql+pymysql://root:110@localhost:3306/test",
                         echo=True,pool_size=5, max_overflow=4,
                         pool_recycle=7200, pool_timeout=30)
```

上述代码只是给出连接 MySQL 的语句，其他数据的连接如表 15-1 所示。

表 15-1　主流数据库连接方式

数据库	连接字符串
Microsoft SQL Server	mssql+pymssql://username:password@ip:port/dbname
MySQL	mysql+pymysql://username:password@ip:port/dbname
Oracle	cx_Oracle://username:password@ip:port/dbname
PostgreSQL	postgresql://username:password@ip:port/dbname
SQLite	sqlite://file_path

15.3 创建数据表

完成数据库的连接后，可以通过 SQLAlchemy 对数据表进行创建和删除，由图 10-2 可知，test 数据库是没有数据表的，使用 SQLAlchemy 创建数据表，代码如下：

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, DateTime
Base = declarative base()

class mytable(Base):
    # 表名
      tablename   = 'mytable'
    # 字段，属性
    id = Column(Integer, primary key=True)
    name = Column(String(50), unique=True)
    age = Column(Integer)
    birth = Column(DateTime)
    class name = Column(String(50))
# 创建数据表
Base.metadata.create_all(engine)
```

引入 declarative_base 模块，生成其对象 Base，再创建一个类 mytable。一般情况下，数据表名和类名是一致的，__tablename__用于定义数据表的名称，可忽略，忽略时默认类名为数据表名。然后创建字段 id、name、age、birth、class_name。最后使用 Base.metadata.create_all(engine)在数据库中创建对应的数据表。

上述是比较常见的创建数据表的方法之一，还有一种创建方法类似 SQL 语句的创建方法：

```
from sqlalchemy import Column, MetaData, ForeignKey, Table
from sqlalchemy.dialects.mysql import (INTEGER, CHAR)
meta = MetaData()
myclass = Table('myclass', meta,
                Column('id', INTEGER, primary key=True),
                Column('name', CHAR(50), ForeignKey(mytable.name)),
                Column('class name', CHAR(50))
                )
# 创建数据表
myclass.create(bind=engine)
```

此创建方法与前面介绍的创建数据表的方法大有不同，代码比较偏向于 SQL 创建数据表的语法，两者引入的模块也各不相同，导致在创建数据表的时候，创建语法也不一致。不过两者实现的功能是一样的，读者可以根据自己的爱好进行选择。一般情况下，前者较有优势，在数据表已经存在的情况下，前者再创建数据表不会报错，后者就会提示已存在数据表的错误信息。

若要删除数据表，则可用以下代码：

```
# 先删除 myclass，后删除 mytable
myclass.drop(bind=engine)
Base.metadata.drop_all(engine)
```

在删除数据表的时候，一定要先删除设有外键的数据表，也就是先删除 myclass 后才能删除 mytable，两者之间涉及外键，这是在数据库中删除数据表的规则。

以下是完整的代码：

```
# 连接数据库
from sqlalchemy import create engine
engine = create engine(
    "mysql+pymysql://root:1990@localhost:3306/test?charset=utf8",
    echo=True)

# 创建数据表方法一
from sqlalchemy import Column, Integer, String, DateTime
from sqlalchemy.ext.declarative import declarative base
Base = declarative base()

class mytable(Base):
    # 表名
      tablename   = 'mytable'
    # 字段，属性
    id = Column(Integer, primary key=True)
    name = Column(String(50), unique=True)
    age = Column(Integer)
    birth = Column(DateTime)
    class name = Column(String(50))

Base.metadata.create all(engine)

# 创建数据表方法二
from sqlalchemy import Column, MetaData, ForeignKey, Table
from sqlalchemy.dialects.mysql import (INTEGER, CHAR)
meta = MetaData()
myclass = Table('myclass', meta,
                Column('id', INTEGER, primary key=True),
                Column('name', CHAR(50), ForeignKey(mytable.name)),
                Column('class name', CHAR(50))
                )
myclass.create(bind=engine)

# 删除数据表
myclass.drop(bind=engine)
Base.metadata.drop_all(engine)
```

注　意

无论数据表是否已经创建，在使用 SQLAlchemy 时一定要对数据表的属性、字段进行类定义。也就是说，无论通过什么方式创建数据表，在使用 SQLAlchemy 的时候，第一步是创建数据库连接，第二步是定义类来映射数据表，类的属性映射数据表的字段。

15.4 添加数据

完成数据表的创建后，下一步对数据表的数据进行操作。首先创建一个会话对象，用于执行 SQL 语句，代码如下：

```
from sqlalchemy.orm import sessionmaker
DBSession = sessionmaker(bind=engine)
session = DBSession()
```

引入 sessionmaker 模块，指明绑定已连接数据库的 engine 对象，生成会话对象 session，该对象用于数据库的增、删、改、查。

一般来说，常用的数据库操作是增、改、查，SQLAlchemy 对这类操作有自身的语法支持。对 10.4 节中创建的数据表添加数据，代码如下：

```
new data = mytable(name='Li Lei',age=10,birth='2017-10-01',
            class name='一年级一班')
session.add(new data)
session.commit()
session.close()
```

要使用 SQLAlchemy 添加数据，必须已经定义 mytable 对象，mytable 是映射数据库里面的 mytable 数据表。然后设置类属性（字段）对应的添加值，将数据绑定在 session 会话中，最后通过 session.commit()来提交到数据中，就完成对数据库的数据添加了。session.close()用于关闭会话，关闭会话不是必要规定，不过为了形成良好的编码规范，最好添加上。

注 意

如果关闭会话放在 session.commit()之前，这个添加语句就是无效的，因为当前的 session 已经被关闭和销毁。所以在使用 session.close()时，要注意编写的位置。

通过 MySQL 工作台可以看到数据表中已成功添加一条数据，如图 15-3 所示。

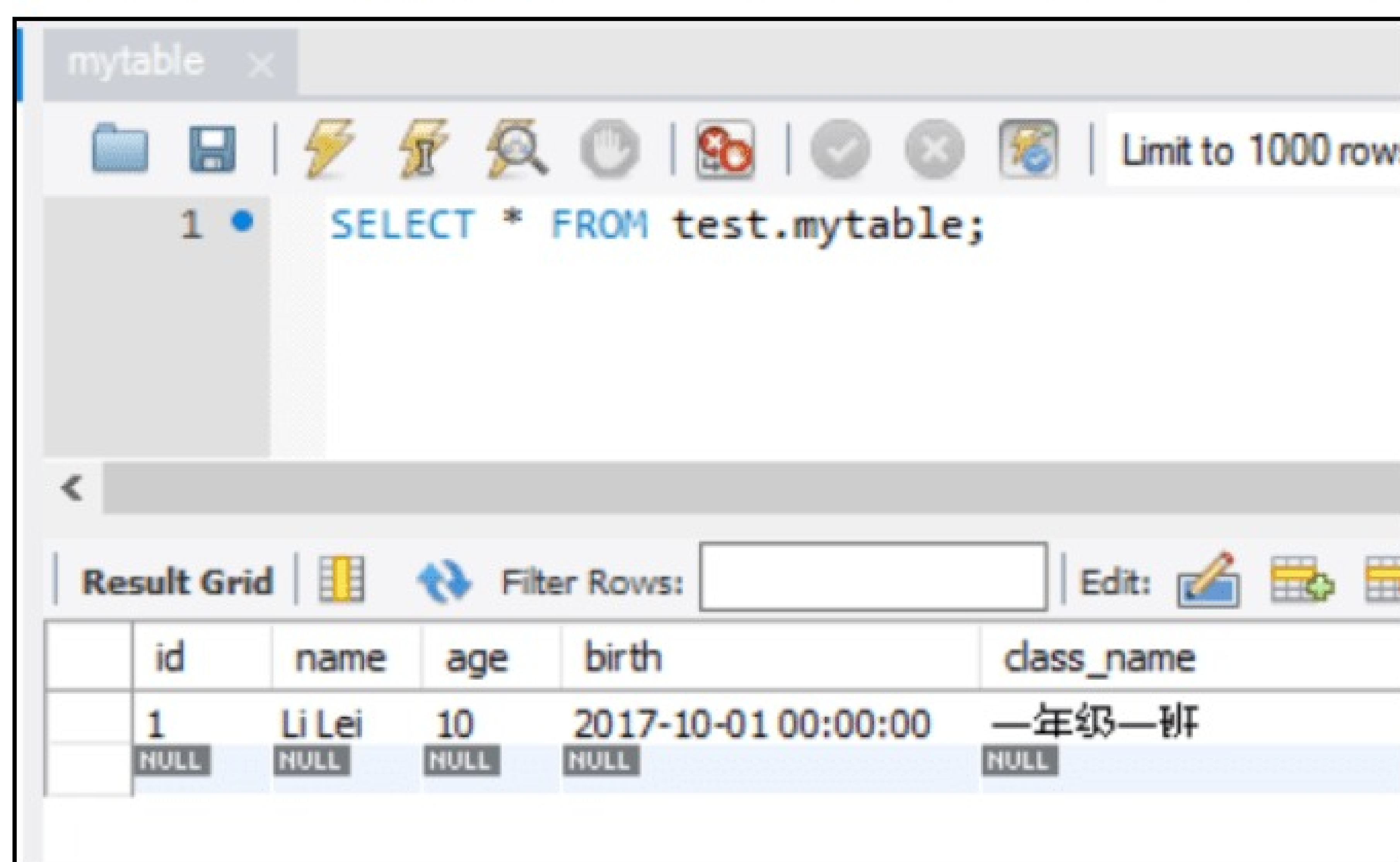


图 15-3　SQLAlchemy 添加数据

15.5　更 新 数 据

目前，数据库中已经添加了一条数据，如果要对这条数据进行更新，SQLAlchemy 提供了以下两种更新数据的方法。

（1）使用 update 方法更新数据，代码如下：

```
session.query(mytable).filter_by(id=1).update({ mytable.age : 12})
session.commit()
session.close()
```

首先查询 mytable 表 id 为 1 的数据；然后使用 update 对这条数据进行更新，update 数据的格式是字典类型，通过键值的方式对数据进行更新；接着使用 session.coomit()执行更新语句；最后使用 session.close()关闭当前会话，释放资源。

如果批量更新，就可以将 filter_by(id=1)去掉，这样能将 mytable 中 age 字段的值全部更新为 12。filter_by 相当于 SQL 语句里面的 where 条件判断。

使用赋值方式更新数据，代码如下：

```
get_data = session.query(mytable).filter_by(id=1).first()
get_data.class_name = '三年级三班'
session.commit()
session.close()
```

使用赋值方式也是将数据查询出来，生成查询对象，然后对该对象的某个属性重新赋值，最后提交到数据库执行。这种方法对批量更新不太友好，常用于单条数据的更新，若要用这种方法实现批量更新，则只能循环每条数据进行赋值更改。但这种方法对性能影响较大，批量更新使用 update()比较合理。

运行结果如图 15-4 所示。

图 15-4　SQLAlchemy 更新数据

15.6 查询数据

SQLAlchemy 对数据库多种查询方式有很好的语法支持。首先对 mytable 和 myclass 加入部分数据，以便更好地讲解，如图 15-5 所示。

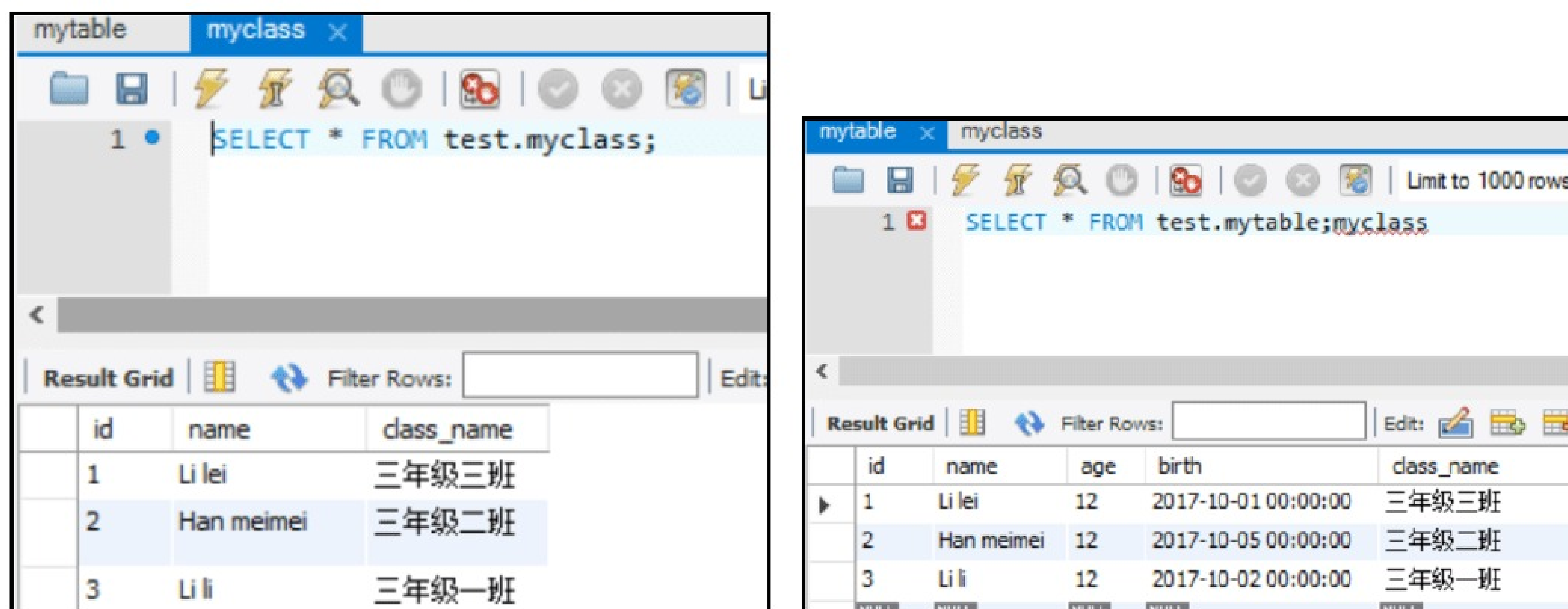


图 15-5　数据表数据内容

由图 15-5 可以看到，两个数据表已添加部分数据，查询某个数据表中数据的代码如下：

```
# 查询 myclass 全部数据
get_data = session.query(myclass).all()
for i in get_data:
    print('我的名字是：' + i.name)
    print('我的班级是：' + i.class_name)
session.close()
```

代码 session.query(myclass)相当于 SQL 语句里面的 select * from myclass; 而 all()是将数据以列表的形式返回。

如果要查询某一字段，如 SQL 语句 select name,class_name from myclass，代码如下：

```
get_data = session.query(myclass.name, myclass.class_name).all()
for i in get_data:
    print('我的名字是：' + i.name)
    print('我的班级是：' + i.class_name)
session.close()
```

设置筛选条件，SQLAlchemy 有两种筛选方法，代码如下：

```
# 根据条件查询某条数据
# 筛选方法一：
# get_data = session.query(myclass).filter(myclass.id==1).all()
# 筛选方法二：
get_data = session.query(myclass).filter_by(id=1).all()
print('数据类型是：' + str(type(get_data)))
for i in get_data:
    print('我的名字是：' + i.name)
    print('我的班级是：' + i.class_name)
```

代码分别有两个 get_data 对象，两者的区别在于 filter 和 filter_by。

（1）字段写法：filter 筛选的字段是带类名（表名）的，而 filter_by 只需筛选字段即可。
（2）判断条件：filter 比 filter_by 多出一个等号。
（3）作用范围：filter 可以用于单表或者多表查询，而 filter_by 只能用于单表查询。

all()方法是将查询数据以列表的形式返回，但只查询一条数据的时候，可以用 first()返回第一条数据。代码如下：

```
get_data = session.query(myclass).filter_by(id=1).first()
print('数据类型是：' + str(type(get_data)))
print('我的名字是：' + get_data.name)
print('我的班级是：' + get_data.class_name)
```

实现多条件筛选，如 SQL 的 select * from mytable where id>1 and class_name='三年级二班'，实现方法如下：

```
get_data = session.query(mytable).filter(mytable.id >= 2,
           mytable.class_name == '三年级二班').first()
print('数据类型是：' + str(type(get_data)))
print('我的名字是：' + get_data.name)
print('我的班级是：' + get_data.class_name)
```

多条件查询只需要在查询条件中添加多个查询内容即可，每个查询内容以英文逗号隔开。如果将 SQL 语句的多条件查询“and”改成“or”，SQLAlchemy 代码如下：

```
from sqlalchemy import or_
session.query(mytable).filter(or_(mytable.id >= 2,
                              mytable.class_name == '三年级二班')).all()
```

如果涉及多表查询的内连接查询和外连接查询，实现代码如下：

```
# 内连接
get_data = session.query(mytable).join(myclass).filter(
                               mytable.class_name == '三年级二班').all()
print('数据类型是：' + str(type(get_data)))
for i in get_data:
       print('我的名字是：' + i.name)
       print('我的班级是：' + i.class_name)
# 外连接
get_data = session.query(mytable).outerjoin(
         myclass).filter(mytable.class_name=='三年级二班').all()
```

代码中的 join 和 outerjoin 与 SQL 语句中的 INNER JOIN 和 FULL OUTER JOIN 意思一致，两者之间在实现功能和性能上存在明显的差别。

一般来说，如果涉及复杂的查询语句，特别涉及多表查询和复杂的查询条件时，SQLAlchemy 还可以直接执行 SQL 语句，代码如下：

```
sql = 'select * from mytable '
session.execute(sql)
# 如果涉及更新、添加数据，就需要 session.commit()
session.commit()
```

15.7 本章小结

本章主要介绍了 ORM 框架的 SQLAlchemy 的功能和使用，SQLAlchemy 的理念是 SQL 数据库的量级和性能重要于对象集合，而对象集合的抽象又重要于表和行。

SQLAlchemy 操作数据库的流程如下。

- 连接数据库：使用 create_engine()实现连接，需了解 create_engine()各个参数的作用。
- 创建数据表：定义实体类映射数据表结构，通过操作类属性从而操作数据表字段。
- 创建持久化对象：引入 sessionmaker 模块，绑定已连接数据库的 engine 对象，生成会话对象 session。
- 添加数据：对实体类的属性赋值，通过 session.add()方法添加数据，通过 session.commit()提交到数据库。
- 使用更新数据：先查询需要修改的数据对象再更新。更新方法有修改对象属性值和使用 update()方法更新数据。
- 查询数据：掌握 SQLAlchemy 查询语句，区分 filter_by 和 filter 的差异，理解多条件查询和多表查询。
- 执行 SQL 语句：SQLAlchemy 使用 execute()方法执行 SQL 语句。

第 16 章 MongoDB 数据库操作

16.1 MongoDB 介绍

MongoDB 是一种基于分布式文件存储的数据库，由 C++语言编写，旨在为 Web 应用提供可扩展的高性能数据存储解决方案。MongoDB 是介于关系数据库和非关系数据库之间的产品，是非关系数据库中功能最丰富、最像关系数据库的数据库。MongoDB 支持的数据结构非常松散，类似于 JSON 的 BSON 格式，因此可以存储比较复杂的数据类型。MongoDB 最大的特点是支持的查询语言非常强大，其语法有点类似面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

MongoDB 的特点是高性能、易部署、易使用，存储数据非常方便。主要功能特性有：

（1）面向集合存储、易存储对象类型的数据。
（2）模式自由。
（3）支持动态查询。
（4）支持完全索引，包含内部对象。
（5）支持查询。
（6）支持复制和故障恢复。
（7）使用高效的二进制数据存储，包括大型对象（如视频等）。
（8）自动处理碎片，以支持云计算层次的扩展性。
（9）支持 Ruby、Python、Java、C++、PHP、C#等多种语言。
（10）文件存储格式为 BSON（一种 JSON 的扩展）。
（11）可通过网络访问。

所谓“面向集合”（Collection-Oriented），意思是数据被分组存储在数据集中，被称为一个集合（Collection）。每个集合在数据库中都有一个唯一的标识名，并且可以包含无限数目的文档。集合的概念类似关系型数据库（RDBMS）里的表（Table），不同的是 MongoDB 不需要定义任何

模式（Schema），具有闪存高速缓存算法，能够快速识别数据库内大数据集中的热数据，提供一致的性能改进。

模式自由（Schema-Free），意味着对于存储在 MongoDB 数据库中的文件，不需要知道它的任何结构定义。如果需要，完全可以把不同结构的文件存储在同一个数据库里。

存储在集合中的文档被存储为键-值对的形式。键用于唯一标识一个文档，为字符串类型，而值则可以是各种复杂的文件类型。我们称这种存储形式为 BSON（Binary Serialized Document Format）。

MongoDB 已经在多个站点部署，其主要场景如下：

（1）网站实时数据处理。非常适合实时地添加、更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性。

（2）缓存。由于性能很高，因此适合作为信息基础设施的缓存层。在系统重启之后，由它搭建的持久化缓存层可以避免下层的数据源过载。

（3）高伸缩性的场景。非常适合由数十或数百台服务器组成的数据库，它的路线图中已经包含对 MapReduce 引擎的内置支持。

16.2 MogoDB 的安装及使用

使用 Python 操作 MongoDB 需要搭建开发环境，本节主要介绍在 Windows 下搭建 Python+MongoDB 环境配置。配置环境需安装 MongoDB、MongoDB 可视化工具和 Python 操作 MongoDB 的第三方库 PyMongo。

16.2.1 MongoDB 的安装与配置

MongoDB 的安装包可在官方网站下载社区版（www.mongodb.com/download-center#community），如图 16-1 所示。



图 16-1　MongoDB 官方下载版本

下载完成之后，直接打开安装包，单击“Next”按钮按提示完成安装即可。完成安装后，进

入 MongoDB 默认安装目录（C:\Program Files\MongoDB\Server\3.4），在当前目录下新建文件夹 data 和 log，分别用于存放数据库文件和 log 日志文件，再创建一个 mongo.conf 配置文件，如图 16-2 所示。



图 16-2　MongoDB 安装目录

打开新创建的 mongo.conf，输入以下代码：

```
# 数据库文件路径
dbpath = C:\Program Files\MongoDB\Server\3.4\data
# 日志输出文件路径
logpath = C:\Program Files\MongoDB\Server\3.4\log\mongo.log
# 错误日志采用追加模式
logappend = true
# 启用日志文件，默认启用
journal = true
# 这个选项可以过滤掉一些无用的日志信息，若需要调试使用，则设置为 false
quiet = true
# 端口号，默认为 27017
port = 27017
```

代码中的数据库文件路径为新建的 data 文件夹路径（data 文件夹的路径没有硬性规定，一般默认为 MongoDB 的安装目录），日志文件路径为新建的 log 文件夹路径。写入配置信息后保存关闭文件，然后打开 CMD 窗口（终端），路径切换到图 16-2 中的 bin 目录，依次输入以下命令：

```
mongod --config "配置文件 mongo.conf 绝对路径" --install --serviceName "MongoDB"
net start MongoDB
```

以上命令代表将 MongoDB 数据库服务器添加到 Windows 服务，这样可免去每次手动开启 MongoDB。运行结果如图 16-3 所示。

```
C:\Program Files\MongoDB\Server\3.4\bin>mongod --config "C:\Program Files\MongoDB\Server\3.4\mongo.conf" --install --ser
viceName "MongoDB"

C:\Program Files\MongoDB\Server\3.4\bin>net start MongoDB
MongoDB 服务正在启动 .
MongoDB 服务已经启动成功。


C:\Program Files\MongoDB\Server\3.4\bin>
```

图 16-3　MongoDB 配置信息

完成配置设置后，在浏览器中输入 http://127.0.0.1:27017/验证配置是否成功，若出现如图 16-4 所示的内容，则说明配置成功。



图 16-4　MongoDB 配置成功

16.2.2　MongoDB 可视化工具

可视化工具可帮助使用者快速查看数据库的使用情况，MongoDB 常用的可视化工具有 RoboMongo 和 MongoBooster。

以 RoboMongo 使用为例，官方网站下载地址为 https://robomongo.org/download，下载后运行.exe 文件，按提示可完成安装。然后运行软件，单击“MongoDB Connections”界面中的“Create”按钮，弹出“Connections Settings”，输入 Name 和 Address 的信息：Name 为对该连接的命名，可自定义命名；Address 处分别输入数据库 IP 地址和端口。此处以本地数据库为例，如图 16-5 所示。

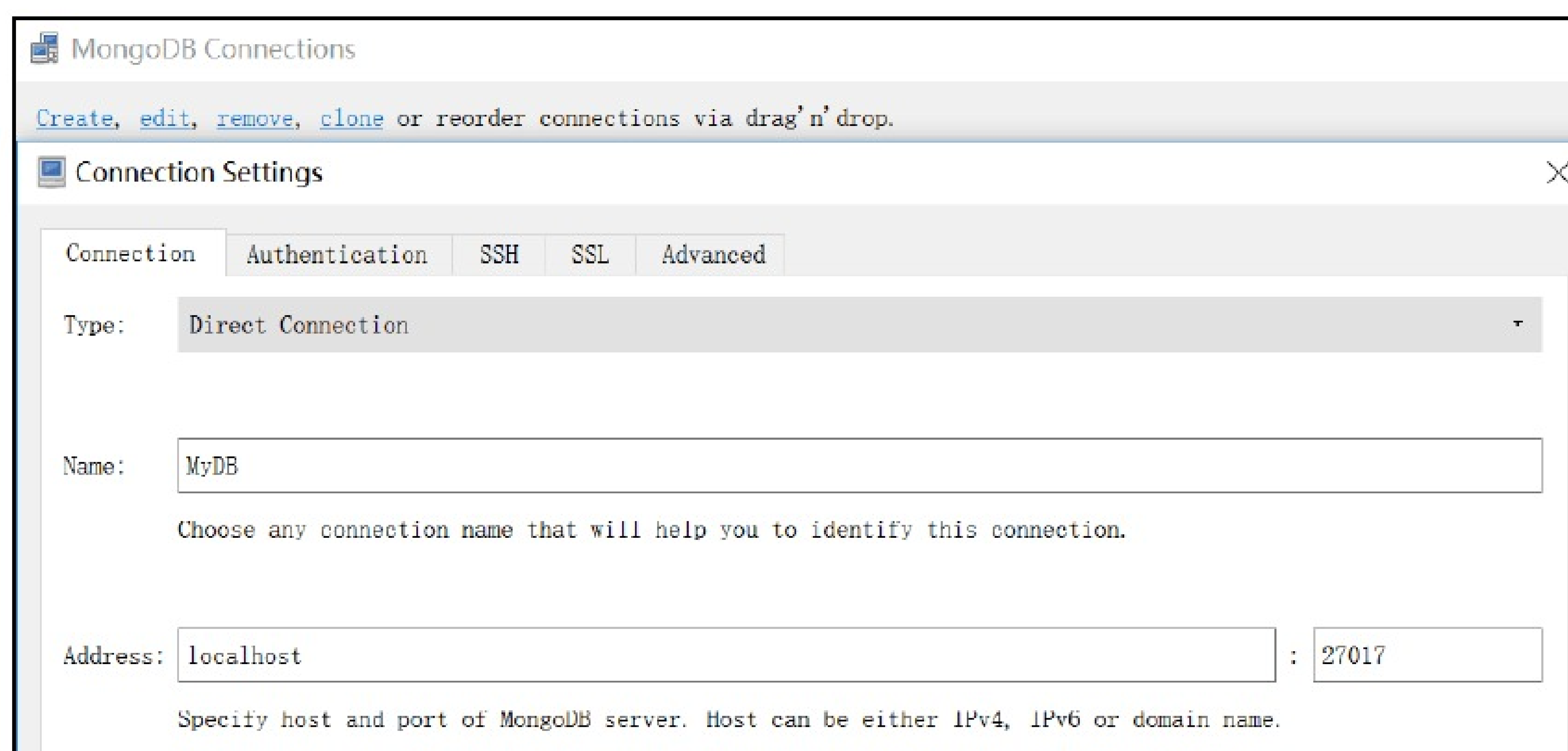


图 16-5　RoboMongo 创建数据库连接

连接数据库后，会看到数据库有一个“system”文件夹，文件夹里有“admin”和“local”数据库，两者皆属于系统数据库，如图 16-6 所示。

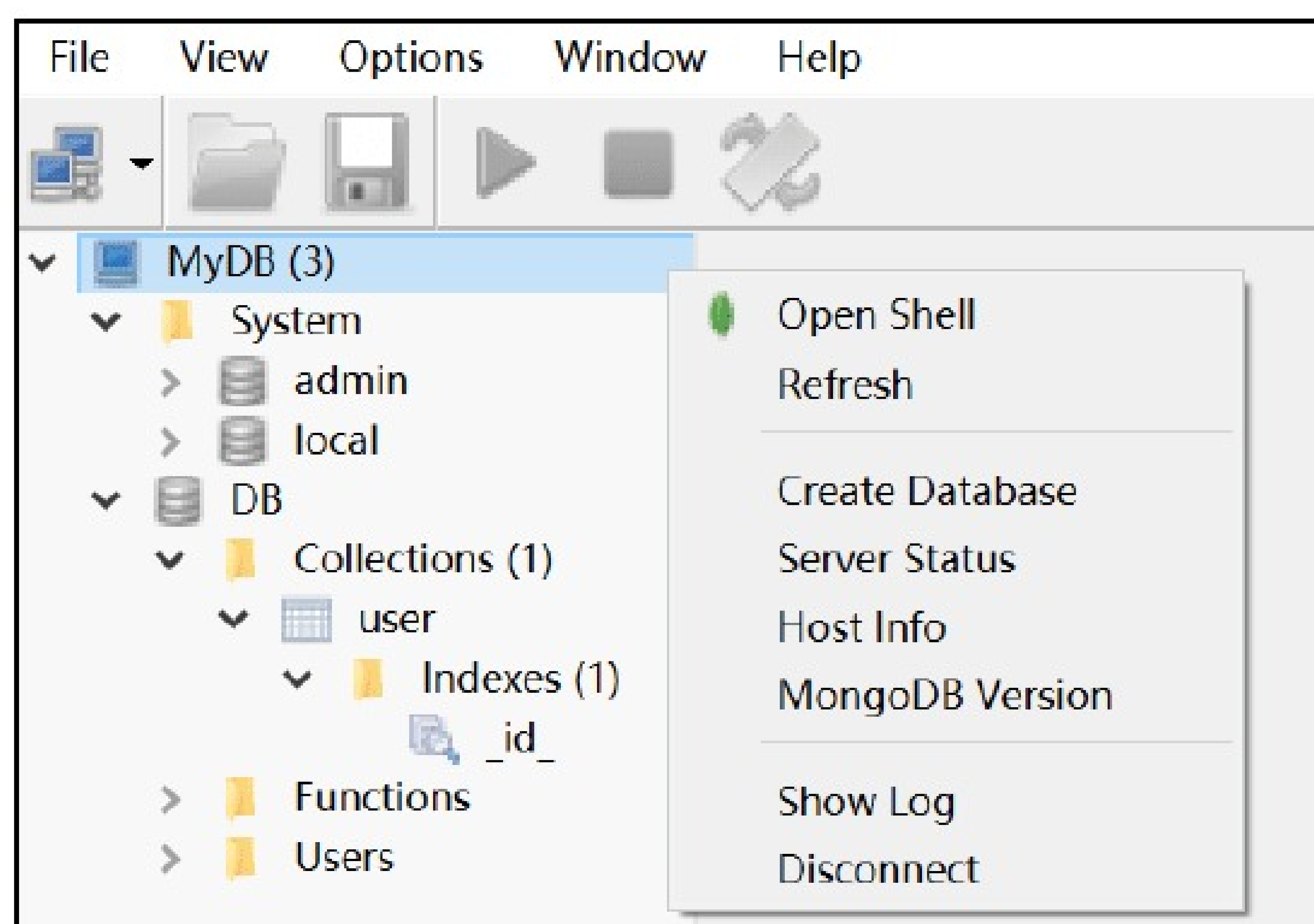


图 16-6　MongoDB 数据结构

结合图 11-6 创建数据库，方法如下：

步骤01 右击“MyDB”，单击“Create Database”，将数据库命名为“DB”。

步骤02 打开数据库“DB”，右击“Collections”，选择“Create Collection”，命名为“user”。新建的 user 称为集合，相当于关系数据库里面的数据表。

步骤03 右击“user”，选择“Insert Document”。Document 代表文档内容，相当于 MySQL 里数据表中的数据。Document 是 BSON 格式，类似 JSON，如图 16-7 所示。

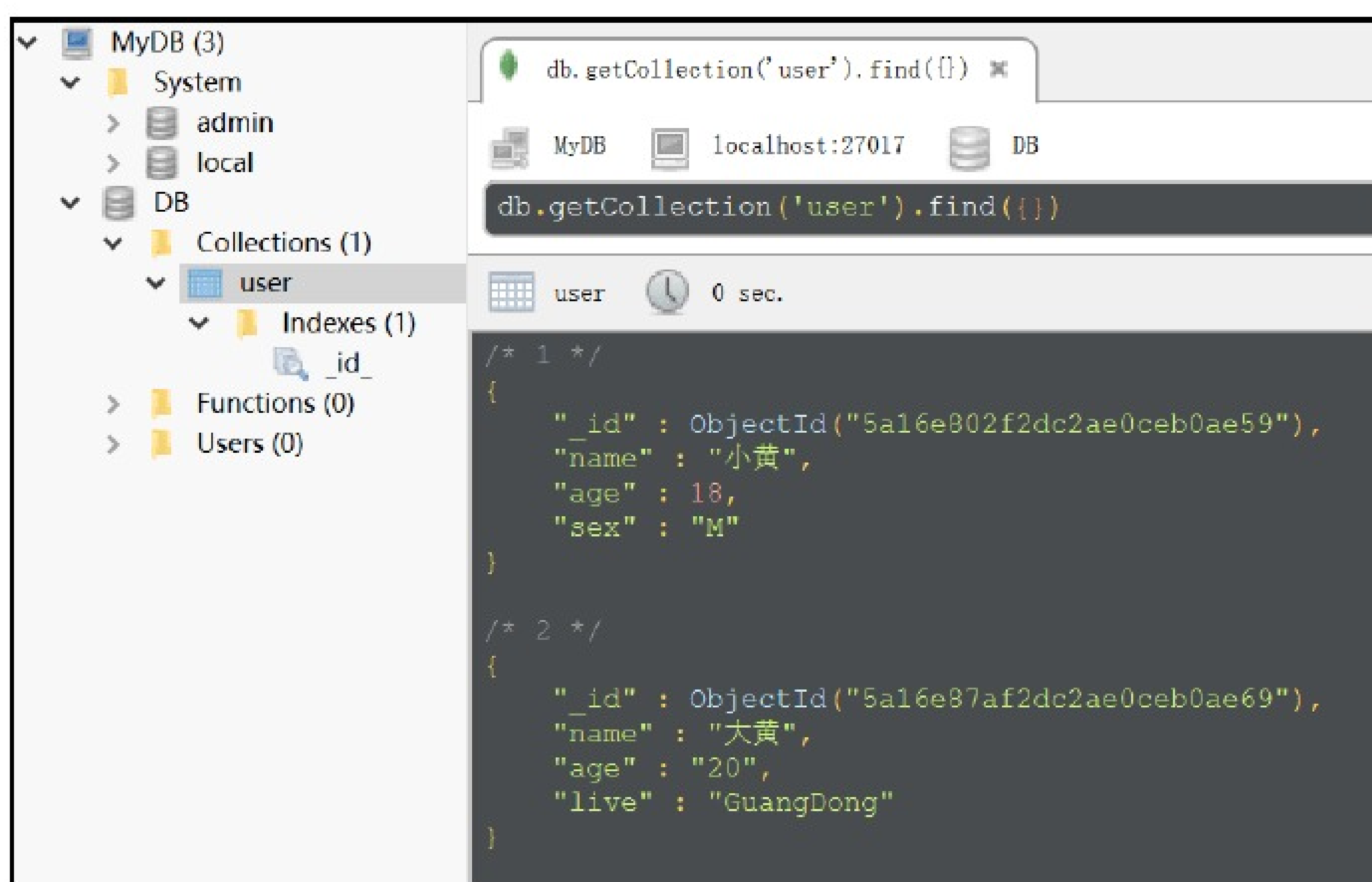


图 16-7　MongoDB 添加文档

步骤04 集合 user 里有文件夹“Indexes”，用于实现集合的索引功能；文件夹“Functions”用于实现脚本功能；在“Users”中设定用户账号密码，用于设置访问权限。

16.2.3　PyMongo 的安装

PyMongo 是 Python 操作 MongoDB 的第三方库，有庞大的社区，功能较为稳定和完善。建议使用 pip 安装 PyMongo：

```
pip install pymongo
```

完成安装后，打开 CMD 窗口，通过导入模块测试是否安装成功：

```
>>> import pymongo
>>> pymongo.  version
'3.7.2'
```

16.3　连接 MongoDB 数据库

通过前面的介绍，相信大家对 MongoDB 的数据结构有了一定的了解，本节介绍 Python 连接 MongoDB 数据库。

使用 Python 实现对 MongoDB 操作的原理与连接关系式数据库一样：连接数据库→访问数据表（集合）→增删改查。

Python 连接 MongoDB 主要由 PyMongo 实现，连接代码如下：

```
import pymongo
# 创建对象，连接本地数据库
# 方法一：
client = pymongo.MongoClient()
# 方法二：
client = pymongo.MongoClient('localhost', 27017)
# 方法三：
client = MongoClient('mongodb://localhost:27017/')
# 连接 DB 数据库
db = client['DB']
# 连接集合 user，集合类似关系数据库的数据表
# 如果集合不存在，就会新建集合 user
user_collection = db.user
# 设置文档格式（文档即我们常说的数据）
```

代码使用三种方法创建数据库（client）对象，localhost 是数据库 IP 地址，27017 是数据库端口，db = client['DB']指向需要连接的数据库，user_collection = db.user 指向 user 集合（相当于关系数据库的数据表）。

如果数据库设置了用户验证，在连接命令上要添加验证信息：

```
import pymongo
# 用户验证方法一
client = pymongo.MongoClient()
db_auth = client.admin
db_auth.authenticate(username, password)
# 连接 DB 数据库
db = client['DB']
# 用户验证方法二
client = MongoClient('mongodb://username:password@localhost:27017/')
# 连接 DB 数据库
db = client['DB']
```

上述代码提供两种验证方式，用户验证实质上是在连接数据库的时候，将数据库用户的账号、密码添加到连接语句上实现验证登录。

16.4 添加文档

在 MongoDB 中，常用的操作有添加文档、更新文档、删除文档和查询文档。文档的数据结构和 JSON 基本一样。所有存储在集合中的数据都是 BSON 格式。BSON 是一种类似 JSON 的二进制形式的存储格式，简称 Binary JSON。

文档添加方式分别有单条添加和批量添加，实现代码如下：

```
import pymongo
import datetime
import re
# 创建对象
client = pymongo.MongoClient()
# 连接 DB 数据库
db = client['DB']
```

```
# 连接集合 user，集合类似关系数据库的数据表
# 如果集合不存在，就会新建集合 user
user collection = db.user
# 设置文档格式（文档即我们常说的数据）
user info = {
    " id": 100,
    "author": "小黄",
    "text": "Python 爬虫开发",
    "tags": ["mongodb", "python", "pymongo"],
    "date": datetime.datetime.utcnow()}

# 使用 insert_one 单条添加文档，inserted_id 获取写入后的 id
# 添加文档时，如果文档尚未包含" id"键，就会自动添加" id"。" id"的值在集合中必须是唯一的
# inserted_id 用于获取添加后的 id，若不需要，则可以去掉
user id = user collection.insert one(user info).inserted id
print ("user id is ", user_id)

#批量添加
user_infos = [{
    " id": 101,
    "author": "小黄",
        "text": "Python 爬虫开发",
        "tags": ["mongodb", "python", "pymongo"],
        "date": datetime.datetime.utcnow()},
 {
    " id": 102,
    "author": "小黄_A",
        "text": "Python 爬虫开发 A",
        "tags": {"db":"Mongodb","lan":"Python","modle":"Pymongo"},
        "date": datetime.datetime.utcnow()},
 ]
# inserted_ids 用于获取添加后的 id，若不需要，则可以直接去掉
user id = user collection.insert many(user infos).inserted ids
print ("user id is ", user_id)
```

代码实现了单条添加和批量添加，单条添加的数据是 user_info，该数据是一个字典数据结构；批量添加的数据是 uesr_infos，该数据是一个字典数据组成的列表。执行数据添加分别由 insert_one 和 insert_many 方法实现。数据添加完成后，使用 inserted_id 和 inserted_ids 可返回添加后所自动生成的 id 内容。

16.5　更 新 文 档

更新文档同样分为单条更新和批量更新，分别由 update()和 update_many()实现。文档更新需要加入操作符。操作符的作用：通常文档只会有一部分要更新，利用原子的更新修改器可以使得这部分更新极为高效。MongoDB 提供了许多原子操作，比如文档的保存、修改、删除等。所谓原子操作，就是要么将这个文档保存到 MongoDB，要么没有保存到 MongoDB，不会出现查询到的文档没有保存完整的情况。更新修改器是一种特殊的键，用来指定复杂的更新操作，比如调整、增加或

者删除键，还可能用于操作数组或者内嵌文档。

下面介绍常用的更新操作符。

- $set：用来指定一个键的值。如果这个键不存在，就创建它；如果存在，就执行更新。
- $unset：从文档中移除指定的键。
- $inc：修改器用来增加已有键的值，或者在键不存在时创建一个键。$inc 就是专门来增加（和减少）数字的，只能用于整数、长整数或双精度浮点数。要是用在其他类型的数据上，就会导致操作失败。
- $rename：操作符可以重命名字段名称，新的字段名称不能和文档中现有的字段名相同。如果文档中存在 A、B 字段，将 B 字段重命名为 A，$rename 会将 A 字段和值移除掉，然后将 B 字段名改为 A。
- $push：如果指定的键已经存在，就会向已有的数组末尾加入一个元素；如果指定的键不存在，就会创建一个新的数组。

如何使用操作符实现更新文档呢？例如更新上述已添加的文档的代码如下：

```
# 更新单条文档
# update(筛选条件,更新内容)。筛选条件为空，默认更新第一条文档
user collection.update(
{},
{"$set":{"author":"小黄","text":"Python 爬虫"}}
)
# 批量更新文档，只要将方法 update 改为 update_many 即可
```

在代码中，user_collection 是 11.4 节的集合 user 对象，方法 update 有两个参数，皆为字典格式：第一个字典为筛选条件，若为空，则默认更新第一条文档；第二个字典以操作符为字典的键，更新的内容以字典格式作为字典的值。

16.6 查询文档

查询文档是使用 find()方法产生一个查询来从 MongoDB 的集合中查询到数据。该方法与其他方法的使用大致相同，使用方法如下：

```
# 查询文档,find({"_id":101}),其中{"_id":101}为查询条件
# 若查询条件为空，则默认查询全部
find value = user collection.find({" id":101})
print(list(find_value))
```

如果要实现多条件查询，就需要使用查询操作符：$and 和$or，使用方法如下：

```
# AND 条件查询
find value = user collection.find({
"$and":[{"_id":101},{"author":"小黄"}]
})
print(list(find value))
# OR 条件查询
find_value = user_collection.find({
```

```
"$or":[{"author":"小黄_A"},{"author":"小黄"}]
})
print(list(find_value))
```

方法 find()传递字典作为查询条件，操作符$and 和$or 作为字典的键，字典的值是列表格式的，列表中的元素以字典形式表示，一个元素代表一个查询条件。

如果要实现大于、小于或者不等于这类比较查询，就需要使用比较查询操作符：$lt（小于）、$lte（小于或等于）、$gt（大于）、$gte（大于或等于）、$in（in，符合范围内）、$nin（not in，范围之外)，使用方法如下：

```
# 如查找 id>100 而<102，即_id=101 的文档
find value = user collection.find({
" id":{"$gt":100,"$lt":102}
})
print(list(find value))
# 查找 id 在[100,101]
find value = user collection.find({
"_id":{"$in":[100,101]}
})
print(list(find_value))
```

比较查询和多条件查询存在明显的差别：

（1）多条件查询以操作符为字典的键，比较查询以字段为字典的键。
（2）多条件查询的值是列表格式的，比较查询的值是字典格式的。

如果使用两者组成一个查询，代码如下：

```
find value = user collection.find({
"$and": [{" id": {"$gt":100,"$lt":102}, {" id": {"$in": [100,101]}}]
})
print(list(find_value))
```

从代码中可以看到，多条件查询操作符$and 作为最外层字典的键，比较查询操作符位于最里层字典。$and 是将每个条件连接起来，主要作用于每个查询条件之间；比较查询操作符（$gt 和$in）使条件按照某个规则成立条件判断，主要作用于每个查询条件里面。

当查询条件不明确某个值的时候，可以使用模糊匹配进行查询。在 MongoDB 中实现模糊匹配需要引用正则表达式，代码如下：

```
# 模糊查询实际上是加入正则表达式实现
# 方法一：
find value = user collection.find({
"author": {"$regex": ".*小.*"}
})
print(list(find value))

#方法二：
regex = re.compile(".*小.*")
find value = user collection.find({
"author": regex
})
print(list(find_value))
```

实现模糊匹配有两种不同的方式，两者都需要引用正则表达式来完成模糊功能。

方法一：使用操作符$regex 作为字典的键，告诉数据库这个查询语句要查找字段 author 中含有“小”的内容。

方法二：re.compile 定义了一个 Pattern 实例，这是正则表达式对象，将其实例作为查询条件的值，同样也是告诉数据库需要查找字段 author 中含有“小”的内容。

我们知道 JSON 可以嵌套多个 JSON，MongoDB 的文档也是如此。当查询文档中某个字段嵌套多个文档时，如何将嵌套里面的文档作为查询条件实现文档查询呢？代码如下：

```
# 查询嵌入/嵌套文档
# 查询字段"tags": {"db":"Mongodb", "lan":"Python", "modle":"Pymongo"}
# 查询嵌套字段,只需要查询嵌套里的某个值即可
find_value = user_collection.find({
"tags.db": "Mongodb"
})
print(list(find_value))
```

字段 tags 的值是一个字典类型的数据，也就是说，文档中 tags 字段的值嵌套了另一个文档，如果查询条件是“db”:“Mongodb”，而“db”属于字段 tags，可通过“tags.db”对其进行定位。如果“db”的值再嵌套一个字典，那么可用相同的方式进行下一步的定位，代码如下：

```
# 查询字段"tags": {"db": {"Mongodb":"NoSql","MySql":"Sql"},
        "lan":"Python", "modle":"Pymongo"}
find_value = user_collection.find({
"tags.db.Mongodb": "NoSql"
})
print(list(find_value))
```

16.7 本章小结

MongoDB 是一个基于分布式文件存储的数据库，旨在为 Web 应用提供可扩展的高性能数据存储解决方案，是介于关系数据库和非关系数据库之间的产品，是非关系数据库中功能最丰富的数据库。在当前的爬虫程序中，如何操作 MongoDB 也成为爬虫程序的重要内容。

在本章中，读者要重点掌握以下内容：

（1）熟悉 MongoDB 安装配置。

（2）理解 MongoDB 数据结构。

（3）掌握 MongoDB 数据库的基本操作方法，包括添加文档、更新文档、查询文档，其中：

- 添加文档分为单条添加和批量添加，分别由 insert_one()和 insert_many()实现。
- 更新文档分为单条更新和批量更新，分别由 update()和 update_many()实现，并且掌握更新操作符的使用。

- 查询文档由 find()实现，掌握比较查询、多条件查询、模糊查询和嵌套查询。

第 17 章

实战：爬取 51Job 招聘信息

17.1 项 目 分 析

本项目主要讲述如何爬取前程无忧的招聘信息，通过设定的关键词与地点来搜索站内的招聘信息并实现数据爬取，这是项目的爬虫功能需求。具体说明如下：

- 在浏览器访问 51Job 官方网站(https://www.51job.com/)，并在搜索框输入关键词“Python”，地点选为“广州”，单击“搜索”按钮进入搜索页。
- 在搜索页中，所有符合条件的职位信息以列表的形式排序并设有分页显示。每条职位信息是一个 URL 地址，通过 URL 地址可以进入该职位的详情页。
- 职位详情页也是数据爬取的页面，爬取的数据信息有：职位名称、企业名称、待遇、福利及职位要求等。

项目的开发工具选择 Requests 模块和 BeautifulSoup 模块实现爬虫开发与数据清洗，数据存储选择 Sqlalchemy 框架，数据库选择 MySQL。

下面介绍本项目的实现流程。

17.2 获取城市编号

项目开发是从搜索页开始，观察搜索页的 URL 地址可以发现，URL 含有关键词“Python”和多个数字编号，这些数字编号可能代表某些搜索条件。为了进一步验证这些数字编号的作用，将关键词保持不变，尝试切换不同的地点，比如地点分别选为“广州”、“北京”和“上海”，如图 17-1 所示。



图 17-1　URL 地址变化情况

根据图 17-1 中的 URL 地址及网页内容的变化，可以总结出两个关键点：

（1）当选择不同地点的时候，网页的 URL 地址都会发生变化，从而导致网页内容随之变化，这说明网页内容是通过网站的服务器后台生成的，并非由 Ajax 实现数据动态渲染。

（2）在 URL 地址的“list”后面，首个数字编号代表职位所在的地点，并且每个地方的数字编号都固定不变。

为了得到全国各个城市的数字编号，在浏览器的开发者工具里，单击“Network”选项卡并刷新搜索页，重新捕捉搜索页的请求信息。查看每个请求信息的响应内容，从中查找每个城市的数字编号，最终在“JS”选项卡下找到全国的城市编号，如图 17-2 所示。

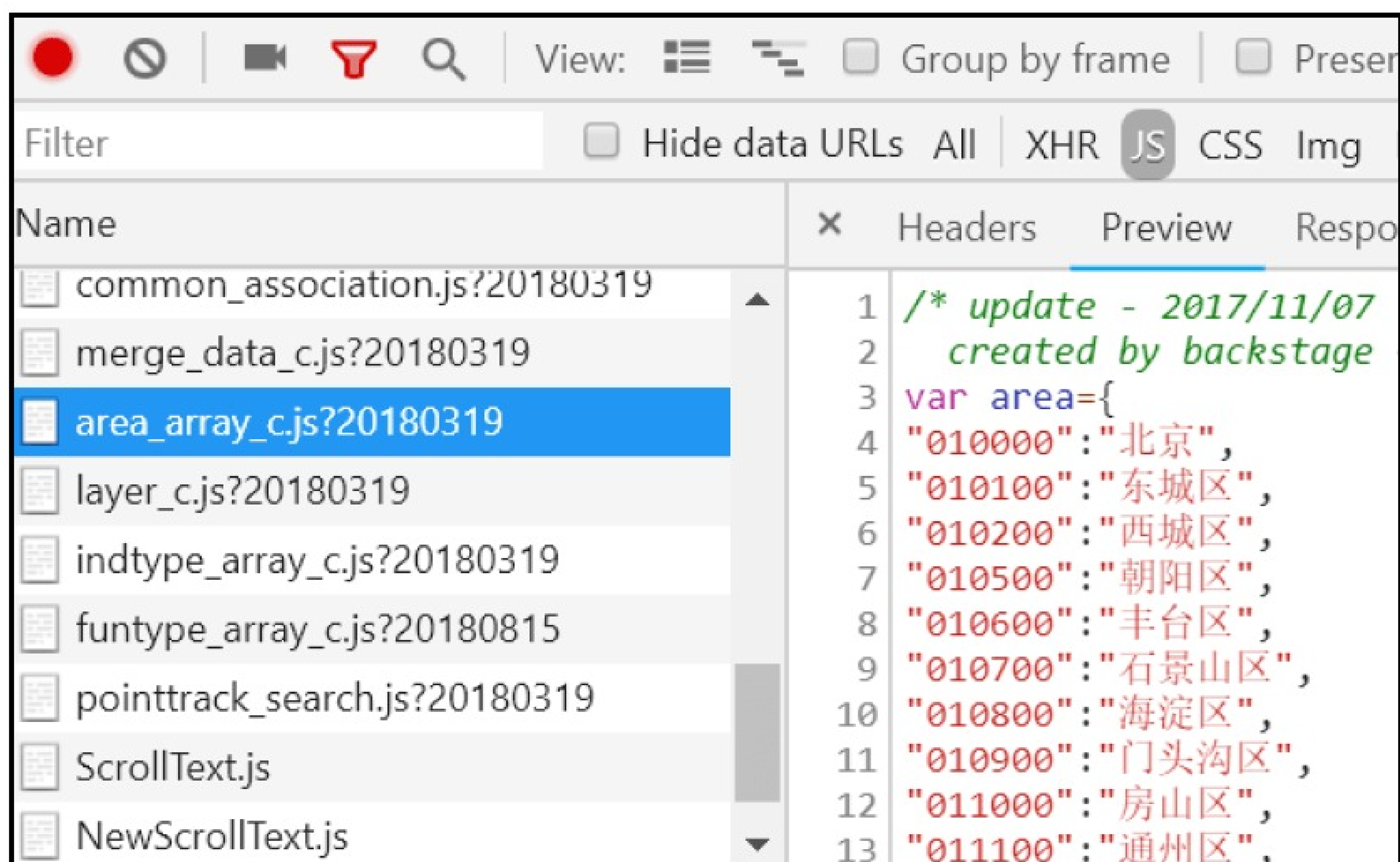


图 17-2　全国的城市编号

由上述分析可知，只要对图 17-2 的请求地址发送 HTTP 请求即可获取每个城市的数字编号，并将该请求的响应内容转换成字典格式，再将字典的键值进行互换。由于爬虫是根据使用者输入城市名来获取相应的数字编号，再通过城市编号构建相应的 URL 地址，所以将字典的键值进行互换可方便数字编号的获取。将城市的数字编号获取功能定义为函数 get_city_code，函数的具体代码如下：

```
import requests
# 获取城市的城市编号
def get_city_code():
    url = 'https://js.51jobcdn.com/in/js/2016/layer/area_array_c.js'
    r = requests.get(url)
     # 字符串转换字典
    city_dict = eval(r.text.split('=')[1].split(';')[0])
     # 字典的键值互换
    city_dict = {v : k for k, v in city_dict.items()}
    return city_dict
```

17.3 获取招聘职位总页数

获取城市编号后，就可以动态构建搜索页的 URL 地址，实现不同地点的不同关键词的职位搜索。在爬取职位信息之前，还需要确定当前职位的总页数，因为同一职位可能会有成千上万条招聘信息，而这些招聘信息都会进行分页处理。

观察搜索页的网页结构可以发现，总页数的获取方式有两种：在分页栏直接获取总页数或者通过总职位数除以每页的职位数。两种方式都可取，但笔者认为后者比前者稍胜一筹，因为考虑到极端情况，当搜索的关键词没有相关的职位，分页栏的总页数显示为 1；而总职位数显示为 0，通过除法计算得出总页数为 0，如图 17-3 所示。



图 17-3　职位信息不存在

总页数的获取若是选择总职位数除以每页的职位数，需要在搜索页上获取总职位数及计算每页的职位数，从而计算得出总页数。每页的职位数在搜索页上通过数数即可获得，每页的职位数量上限为 50；总职位数可以在“Doc”选项卡里找到相应的位置，如图 17-4 所示。

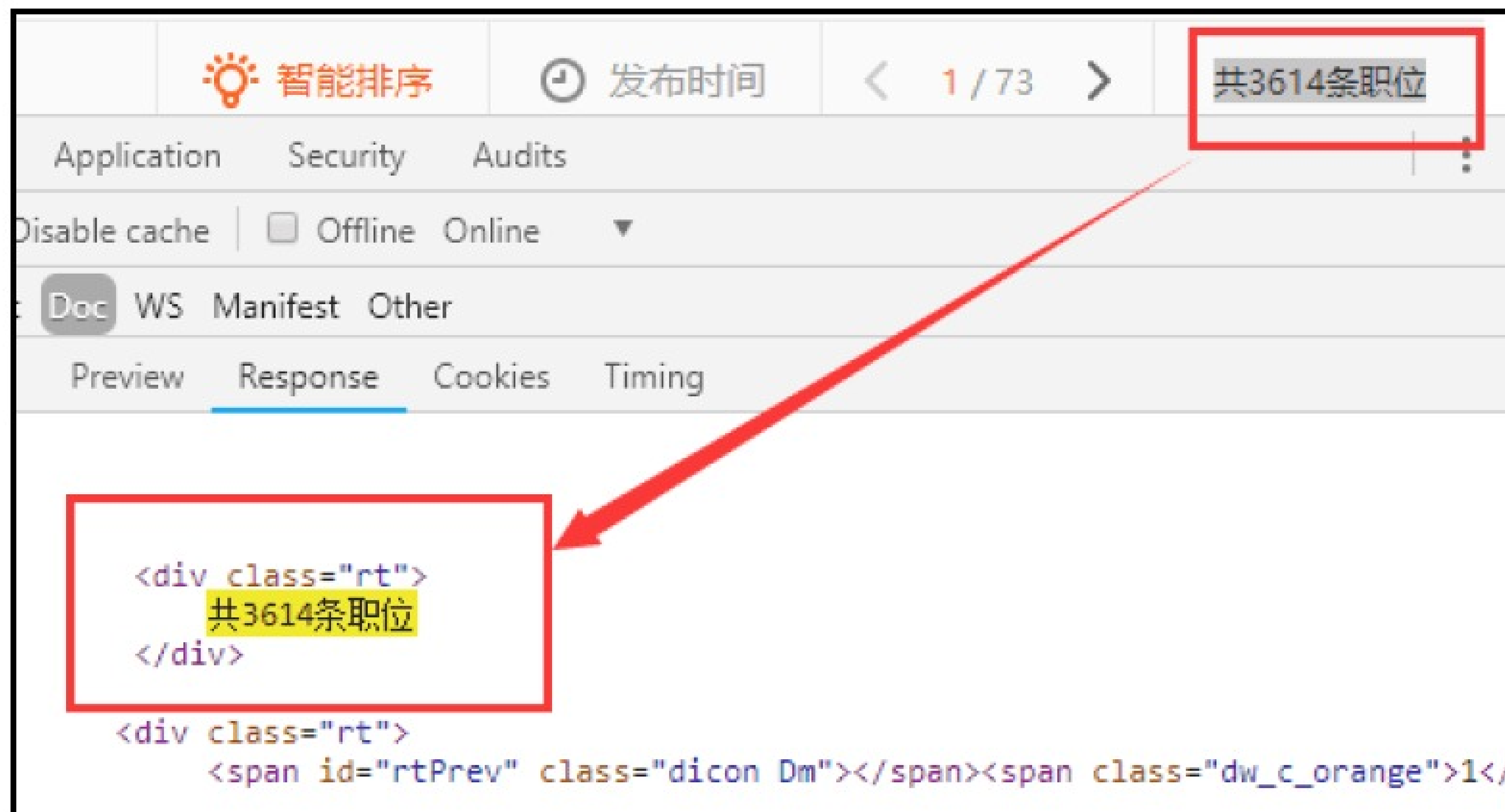


图 17-4　查找总职位数

综合上述分析，我们将职位总页数的获取定义为函数 get_pageNumber，参数为 city_code 和 keyword，分别代表城市编号和关键词，函数代码如下：

```
import requests
from bs4 import BeautifulSoup
import re,math
# 定义请求头
headers = {
    'User-Agent':'Mozilla/5.0 (Windows NT 6.3; Win64; x64)
                  AppleWebKit/537.36 (KHTML, like Gecko)
                  Chrome/66.0.3359.117 Safari/537.36',
    'Host':'search.51job.com',
    'Upgrade-Insecure-Requests': '1'
}
# 获取职位的总页数，参数 city_code 是城市编号，keyword 是职位关键字
def get_pageNumber(city_code, keyword):
    # 获取搜索页的网页信息
    url = 'https://search.51job.com/list/'+str(city_code)+
          ',000000,0000,00,9,99,'+str(keyword)+',2,1.html'
    r = requests.get(url, headers=headers)
    # 使用 BeautifulSoup 进行数据清洗
    soup = BeautifulSoup(r.content.decode('gbk'), 'html5lib')
    # 查找总职位数
    find_page = soup.find('div', class_='rt').getText()
    # 通过正则正则表达式提取数值
    temp = re.findall(r"\d+\.?\d*", find_page)
    # 计算总页数并返回结果
    if temp:
        pageNumber = math.ceil(int(temp[0])/50)
        return pageNumber
    else:
        return 0
```

函数 get_pageNumber 的实现逻辑大致如下：

（1）首先向网站发送 HTTP 请求并获取相应的响应内容，发送请求的 URL 地址为搜索页的 URL 地址，而搜索页的 URL 地址是通过参数 city_code 和 keyword 构建而成。

（2）然后从响应内容中提取总职位数，提取方式由 BeautifulSoup4 模块和 re 模块实现，前者用于对总职位数进行精准定位，后者用来去除总职位数的中文内容。

（3）最后将总职位数和每页的职位数进行除法计算得出总页数，计算过程由 math.ceil 方法实现，ceil 方法是将计算结果的小数点去除并对整数的个位进一。

17.4 爬取每个职位信息

本节实现遍历搜索页的全部职位，从而实现每个职位的信息爬取。整个功能涉及两个遍历循环：遍历总页数和遍历每页的职位信息，具体说明如下。

- 遍历总页数：通过函数 get_pageNumber 获取总页数并对总页数进行遍历处理。每次遍历需要重新构建搜索页的 URL 地址，使当前遍历的次数对应搜索页的页数。构建后的 URL 地址发送 HTTP 请求并从响应内容提取当前页面的所有职位信息。
- 遍历每页的职位信息：对当前搜索页的所有职位的 URL 地址进行遍历访问，通过发送 HTTP 请求进入每个职位的详情页，在职位详情页里爬取目标数据。

在实现总页数的遍历功能之前，需要分析网页的设计结构。在搜索页上分别单击分页栏的不同页数，观察 URL 地址的变化情况，发现 URL 地址某个数字编号会随着页数的变化而变化，如图 17-5 所示。



图 17-5　URL 地址的变化情况

从图 17-5 上的变化情况可知，关键词“Python”后面第二个数字代表页数，只要动态改变这个数值即可得到不同页数的 URL 地址，通过这些 URL 地址可以得到相应的职位信息。以某页的搜索页面为例，在开发者工具的“Doc”选项卡里查找职位详情页的 URL 地址，其 HTML 格式如图 17-6 所示。

在“Doc”选项卡的“Response”里查找某个职位信息可以发现，职位详情页的 URL 地址是在标签 a，但标签 a 没有特殊的属性值，因此直接对标签 a 定位存在一定的难度。接着再看标签 a 的上级标签 p，其属性 class 的属性值为 t1，通过分析属性值 t1 得知，整个网页内容共有 50 个这样的标签 p，也就是说职位信息可以通过标签 p 定位，再由标签 p 定位到标签 a。

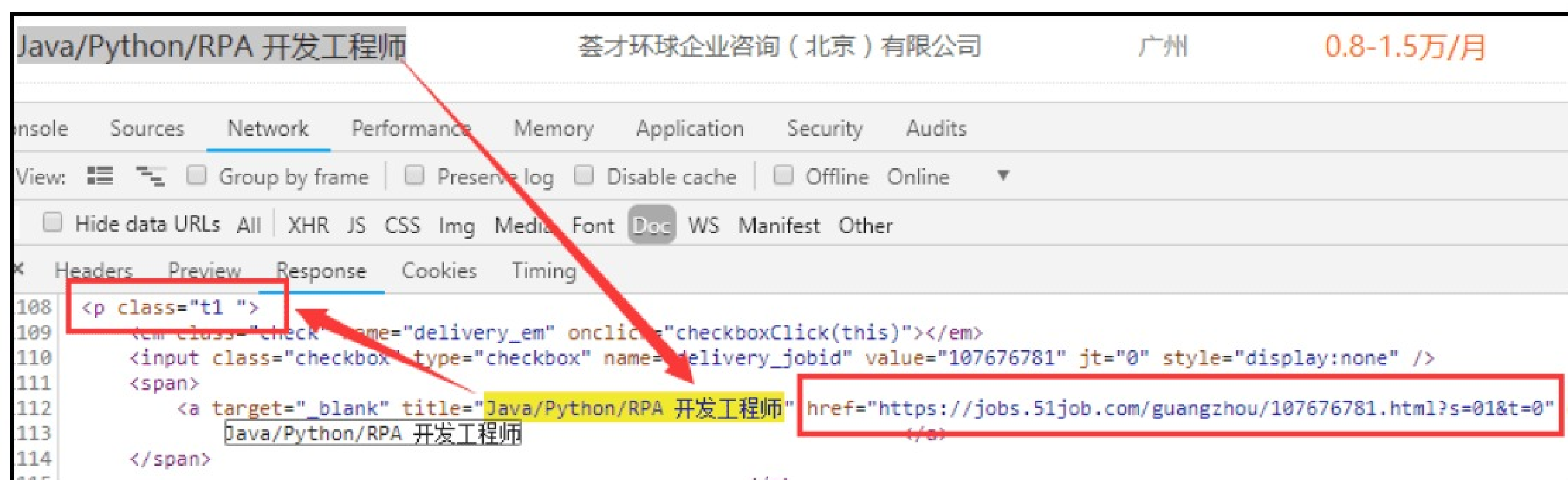


图 17-6　查找职位详情页的 URL 地址

根据搜索页的 URL 地址构成规律以及职位信息的定位方式，将总页数的遍历功能定义为函数 get_page，函数参数为 keyword 和 pageNumber，分别代表关键词和总页数，实现代码如下：

```
# 遍历每一页的职位信息
def get page(keyword, pageNumber):
    for p in range(int(pageNumber)):
        url = 'https://search.51job.com/list/' + str(city_code)+
              ',000000,0000,00,9,99,' + str(keyword) + ',2,' +
              str(p + 1) + '.html'
        r = requests.get(url, headers=headers)
        soup = BeautifulSoup(r.content.decode('gbk'), 'html5lib')
        find p = soup.find all('p', class =re.compile('t1'))
        # 进入职位详情页获取数据
        for i in find p:
            try:
                info dict = None
                print(i.find('a')['href'])
                # 调用函数 get_info
                url = i.find('a')['href']
                info_dict = get_info(url)
                # 入库处理
                if info dict:
                    insert_db(info_dict)
            except Exception as e:
                print(e)
                time.sleep(5)
```

整个函数结构设有两个 for 循环，并且第二个循环设置了 try…except 机制，函数说明如下：

（1）第一个循环是遍历总页数，每次遍历都会构建相应的 URL 地址，并从 URL 地址所对应的网页内容提取所有职位信息。网页内容使用 GBK 编码格式，读者可在“Doc”选项卡的“Response”查看标签<head>即可得知网页的编码格式。

（2）第二次循环是遍历当前搜索页的职位信息，从中提取职位详情页的 URL 地址并赋值给变量 url。

（3）变量 url 以函数参数的形式传递给函数 get_info，函数 get_info 是实现职位详情页的信息爬取，爬取结果以字典的形式表示并赋值给变量 info_dict。

（4）变量 info_dict 作为函数 insert_db 的参数，函数 insert_db 是实现数据入库处理。

（5）try…except 机制是预防函数 get_info 在爬取数据过程中出现异常，因为有些企业的职位详情页比较特殊，比如 http://yumchina.zhiye.com/。

在上述的分析过程中，函数 get_info 是实现职位详情页的信息爬取。在实现函数 get_info 的功能之前，需要确定数据爬取的位置以及分析职位详情页的代码结构。以某个职位信息为例，爬取的目标数据如图 17-7 所示。



图 17-7　爬取的目标数据

爬取的目标数据分为企业信息和职位信息，如职位名、职位要求、企业福利、职位薪资、工作地点以及企业规模等。在浏览器的开发者工具的“Doc”选项卡可以找到这些信息所在位置。由于爬取数据较多，本书只列出部分数据的 HTML 代码，如图 17-8 所示。

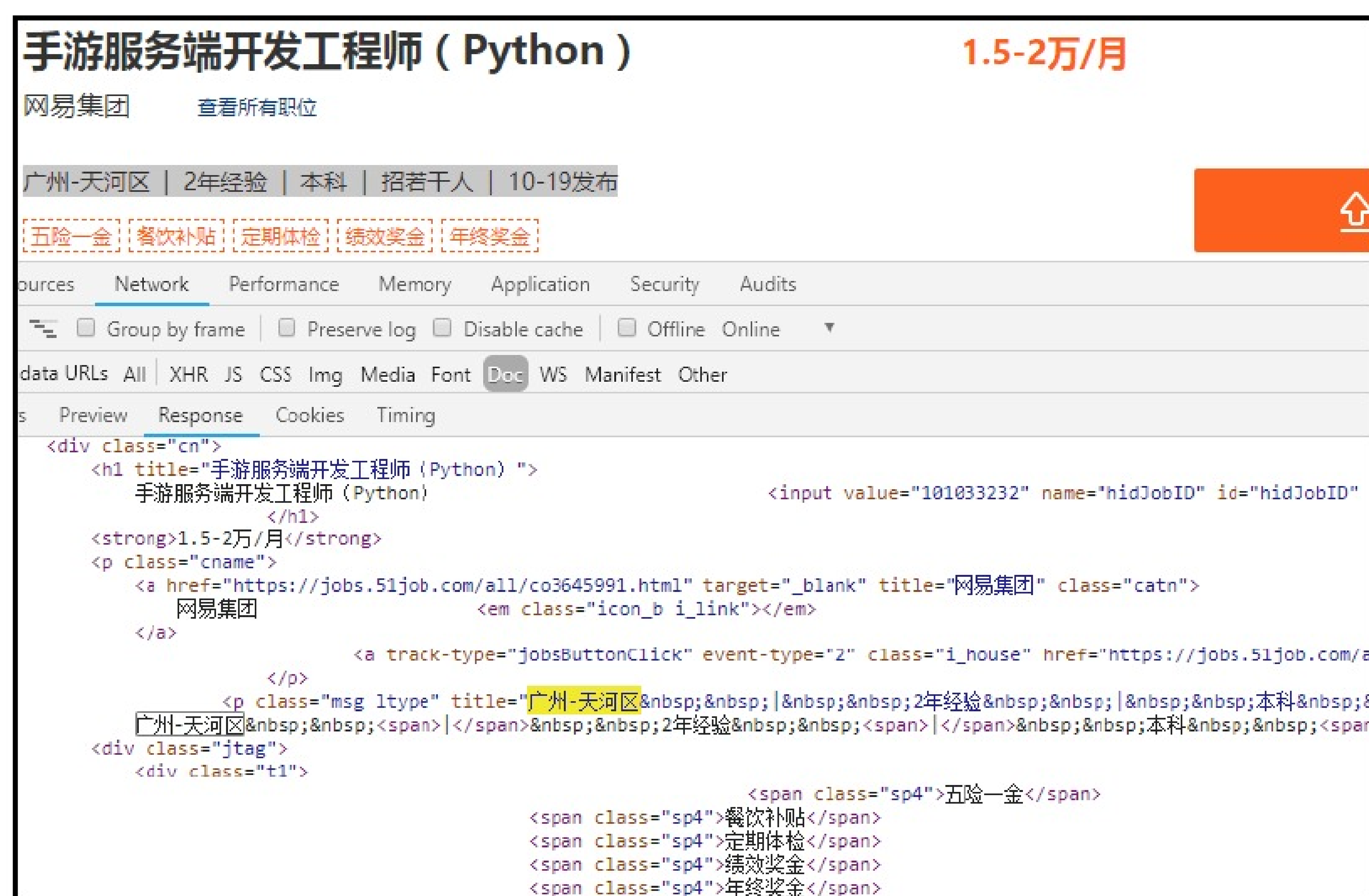


图 17-8　目标数据的 HTML 代码

通过对所有目标数据的 HTML 代码进行分析，发现这些数据可以通过 class 属性实现定位，从而爬取数据内容，实现的代码如下所示：

```
# 爬取职位详情页的数据，参数 url 是职位详情页的链接
def get_info(url):
    temp_dict = {}
    if 'https://jobs.51job.com' in url:
        r = requests.get(url, headers=headers)
        time.sleep(1.5)
        soup = BeautifulSoup(r.content.decode('gbk'), 'html5lib')
        # 职位 ID
        temp_dict['job_id'] = url.split('.html')[0].split('/')[-1]
        # 企业名
        temp_dict['company_name'] = soup.find('a',class_='catn').
                                    getText().strip()
        # 企业类型、规模、经营范围
        com_tag = soup.find('div', class_='com_tag').find_all('p')
        for i in com_tag:
            if 'i_flag' in str(i):
                temp_dict['company_type'] = i.getText()
            if 'i_people' in str(i):
                temp_dict['company_scale'] = i.getText()
            if 'i_trade' in str(i):
                temp_dict['company_trade'] = i.getText()
        # 职位名称
        temp_dict['job_name'] = soup.find('h1').getText().strip()
        # 职位薪资
        temp_dict['job_pay'] = soup.find('div', class_='cn').
                               find('strong').getText().strip()
        # 职位要求：工龄、招聘人数、发布日期、学历要求
        msgltype = soup.find('p', class_='msg ltype').getText().split('|')
        education = ['初中','中专','中技','大专','高中','本科','硕士','博士']
        if msgltype:
            for i in msgltype:
                if '经验' in i.strip():
                    temp_dict['job_years'] = i.strip()
                elif '人' in i.strip():
                    temp_dict['job_member'] = i.strip()
                elif '发布' in i.strip():
                    temp_dict['job_date'] = i.strip()
                elif i.strip() in education:
                    temp_dict['job_education'] = i.strip()
        # 企业福利待遇
        t1 = soup.find('div', class_='t1').find_all('span')
        welfare = []
        for i in t1:
            welfare.append(i.getText().strip())
        temp_dict['company_welfare'] = '/'.join(welfare)
        # 上班地点
        bmsg = soup.find('div', class_='bmsg inbox')
        if bmsg:
            if bmsg.find('p', class_="fp"):
                temp_dict['job_location'] = bmsg.find('p', class_="fp").
                                            getText().strip()
```

```
        # 职位的工作描述
        find describe = soup.find('div', class ='bmsg job msg inbox')
        temp = str(find describe).split('<div class="mt10">')[0]
        Mysoup = BeautifulSoup(temp, 'html5lib')
        temp dict['job describe'] = Mysoup.getText().strip()
        # 招聘来源
        temp_dict['recruit_sources'] = '前程无忧'
        return temp_dict
```

17.5 数 据 存 储

在上一节中提到函数 insert_db，使用该参数实现数据入库处理。为了区分数据爬取与数据入库，我们将数据入库的代码编写在 Insql.py 文件中。数据入库使用 SQLAlchemy 框架+MySQL 数据库实现，在 MySQL 里创建数据库 spiderdb，将数据库编码设为 utf8mb4。然后根据函数 get_info 的返回值来定义数据模型，再通过数据模型在数据库中创建数据表。数据模型的定义如下所示：

```
import time
from sqlalchemy import *
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative base
from sqlalchemy.dialects.mysql import *
engine = create engine(
         'mysql+pymysql://root:1234@localhost/spiderdb?charset=utf8')
DBSession = sessionmaker(bind=engine)
SQLsession = DBSession()
Base = declarative base()

# 定义数据模型，映射数据表
class table info(Base):
      tablename   = 'job info'
    id = Column(Integer(), primary key=True)
    job_id = Column(String(100), comment='职位 ID')
    company name = Column(String(100), comment='企业名称')
    company_type = Column(String(100), comment='企业类型')
    company_scale = Column(String(100), comment='企业规模')
    company trade = Column(String(100), comment='企业经营范围')
    company_welfare = Column(String(1000), comment='企业福利')
    job name = Column(String(3000), comment='职位名称')
    job_pay = Column(String(100), comment='职位薪酬')
    job years = Column(String(100), comment='工龄要求')
    job_education = Column(String(100), comment='学历要求')
    job_member = Column(String(100), comment='招聘人数')
    job_location = Column(String(3000), comment='上班地址')
    job_describe = Column(Text, comment='工作描述')
    job date = Column(String(100), comment='发布日期')
    recruit_sources = Column(String(100), comment='招聘来源')
    log date = Column(String(100), comment='记录日期')
# 创建数据表
Base.metadata.create_all(engine)
```

完成数据模型的定义后，接下来实现函数 insert_db 的功能代码。函数 insert_db 会对职位 ID 进行判断，如果数据表已有这条数据，则对数据表的数据进行更新，否则在数据表中新增数据，函数的代码如下：

```
# 写入数据库
def insert_db(info_dict):
    temp_id = info_dict['job_id']
    # 判断是否已存在记录
    info = SQLsession.query(table_info).filter_by(job_id=temp_id).first()
     # 若存在，更新数据
    if info:
        info.job_id = info_dict.get('job_id','')
        info.company_name = info_dict.get('company_name','')
        info.company_type = info_dict.get('company_type','')
        info.company_trade = info_dict.get('company_trade', '')
        info.company_scale = info_dict.get('company_scale','')
        info.company_welfare = info_dict.get('company_welfare','')
        info.job_name = info_dict.get('job_name', '')
        info.job_pay = info_dict.get('job_pay', '')
        info.job_years = info_dict.get('job_years', '')
        info.job_education = info_dict.get('job_education', '')
        info.job_member = info_dict.get('job_member', '')
        info.job_location = info_dict.get('job_location', '')
        info.job_describe = info_dict.get('job_describe', '')
        info.recruit_sources = info_dict.get('recruit_sources', '')
        info.job_date = info_dict.get('job_date', '')
        info.log_date = time.strftime('%Y-%m-%d',time.localtime(time.time()))
     # 不存在则新增数据
    else:
        inset_data = table_info(
        job_id = info_dict.get('job_id',''),
        company_name=info_dict.get('company_name',''),
        company_type=info_dict.get('company_type',''),
        company_trade=info_dict.get('company_trade',''),
        company_scale=info_dict.get('company_scale',''),
        company_welfare=info_dict.get('company_welfare',''),
        job_name=info_dict.get('job_name', ''),
        job_pay=info_dict.get('job_pay', ''),
        job_years=info_dict.get('job_years', ''),
        job_education=info_dict.get('job_education', ''),
        job_member=info_dict.get('job_member', ''),
        job_location=info_dict.get('job_location', ''),
        job_describe=info_dict.get('job_describe', ''),
        job_date=info_dict.get('job_date', ''),
        recruit_sources=info_dict.get('recruit_sources', ''),
        log_date=time.strftime('%Y-%m-%d', time.localtime(time.time()))
        )
        SQLsession.add(inset_data)
    SQLsession.commit()
```

17.6 爬虫配置文件

从 17.2 到 17.4 节，共定义了 4 个函数，分别是 get_city_code()、get_pageNumber()、get_info() 和 get_page()。每个函数实现的功能说明如下。

- get_city_code()：获取城市编号，可通过城市名转换城市编号来构建搜索页的 URL 地址。
- get_pageNumber()：在搜索页获取总职位数，并通过总职位数除以每页的职位数来获取总页数。函数参数为 city_code 和 keyword，分别代表城市编号和职位关键词。
- get_info()：爬取职位详情页的数据，并以字典的形式返回。函数参数 url 是职位详情页的 URL 地址。
- get_page()：遍历总页数，每次遍历是为爬取当前搜索页所有招聘职位的详情页地址，将爬取的详情页地址作为函数 get_info()的函数参数，通过调用函数 get_info()实现招聘信息的爬取，最后再调用函数 insert_db 进行入库处理。

整个项目的核心功能是由这 4 个爬虫函数及入库函数 insert_db 共同实现，4 个函数的代码编写在爬虫文件 51job.py 中。在本节中，将对核心功能进行包装处理，这也是项目的收尾处理。一般情况下，项目的使用者大多数都是非技术人员，在运行程序之前，使用者需要设置关键词和地点，如果使用者在源码里设置相关参数，由于使用者不懂技术，很容易对源码造成破坏，导致程序无法运行。

因此可以为程序添加配置文件，使用者只需对配置文件的配置信息进行修改即可，无需改动源码，这样可以防止源码修改而引发的异常问题。配置文件的文件扩展名为 conf，它可支持 Windows、Linux 和 MacOS 系统。在爬虫文件 51job.py 的同一目录下新增配置文件 51job.conf，并以记事本的方式打开配置文件，在文件里编写以下配置信息：

```
[51job]
keyword = python,java
city = 广州,北京,上海
```

在配置信息中，“[51job]”是配置信息的标题，一个配置文件可以设有多个这样的标题，它的作用是方便程序对配置信息的定位与查找。本项目设计的爬虫程序是根据上述两个条件进行数据爬取，所以在标题下分别设有关键词和地点。每条配置信息（即关键词和地点）可以设置多个配置内容，每个配置内容之间使用英文逗号隔开。

如果设置了多个配置内容，那么爬虫的爬取方式是根据两个配置内容的组合方式进行爬取，配置信息的内容越多，产生的组合越多。比如“python-广州”、“python-北京”、“python-上海”、“java-广州”、“java-北京”、“java-上海”……以此类推。

完成配置文件 51job.conf 的配置后，还需要对爬虫文件 51job.py 进行修改。爬虫文件 51job.py 只是定义了 4 个爬虫函数，还需要实现配置文件的读取和程序的运行方式，具体的代码如下：

```
if __name__ =='__main__':
    # 读取同一路径的配置文件
    cf = configparser.ConfigParser()
    cf.read("51job.conf")
```

```
    keyword = str(cf.get('51job', 'keyword')).split(',')
    city = str(cf.get('51job', 'city')).split(',')

    # 程序的运行方式
    for c in city:
        # 获取城市编号
        city_code = get_city_code()[c]
        for k in keyword:
            # 获取总页数
            pageNumber = get_pageNumber(city_code, k)
            # 遍历总页数
            get_page(k, pageNumber)
```

配置文件的读取是由 Python 的标准库 configparser 实现，使用 configparser 模块读取配置信息并将配置信息进行分段截取，使每个配置内容（即关键词和地点）以列表的元素表示。通过遍历列表 keyword 和 city 即可实现不同条件组合的数据爬取。

由于爬虫文件 51job.py 的代码篇幅较长，本书在讲述过程中只列出相应的代码，读者可能对整个爬虫程序缺乏整体认知，笔者建议读者通过源码文件（源码可在本书前言的地址下载）进行学习与总结。

17.7　本 章 小 结

本章项目主要讲述如何爬取前程无忧的招聘信息，通过设定的关键词与地点来搜索站内的招聘信息并实现数据爬取，这是项目的爬虫功能需求。具体说明如下：

（1）在浏览器访问 51Job 官方网站（https://www.51job.com/），并在搜索框输入关键词“Python”，地点选为“广州”，单击“搜索”按钮进入搜索页。

（2）在搜索页中，所有符合条件的职位信息以列表的形式排序并设有分页显示。每条职位信息是一个 URL 地址，通过 URL 地址可以进入该职位的详情页。

（3）职位详情页也是数据爬取的页面，爬取的数据信息有：职位名称、企业名称、待遇、福利以及职位要求等。

项目定义了 4 个爬虫函数和一个入库函数，分别是 get_city_code()、get_pageNumber()、get_info()、get_page()。每个函数实现的功能说明如下。

- get_city_code()：获取城市编号，可通过城市名转换城市编号来构建搜索页的 URL 地址。
- get_pageNumber()：在搜索页获取总职位数，并通过总职位数除以每页的职位数来获取总页数。函数参数为 city_code 和 keyword，分别代表城市编号和职位关键词。
- get_info()：爬取职位详情页的数据，并以字典的形式返回。函数参数 url 是职位详情页的 URL 地址。
- get_page()：遍历总页数，每次遍历是爬取当前搜索页所有招聘职位的详情页地址，将爬取的详情页地址作为函数 get_info()的函数参数，通过调用函数 get_info()实现招聘信息的爬取，最后再调用函数 insert_db 进行入库处理。

- insert_db()：对职位 ID 进行判断，如果数据表已有这条数据，则对数据表的数据进行更新，否则在数据表中新增数据。

项目的开发工具选择 Requests、BeautifulSoup 模块和 SQLalchemy 框架实现，读者可以尝试使用 Requests-HTML 和 Sqlalchemy 框架改写上述爬虫程序。

第 18 章

实战：分布式爬虫——QQ 音乐

18.1 项目分析

现在的音乐类网站仅提供歌曲在线免费试听，如果下载歌曲，往往要开通会员或者收取版权费用，但通过爬虫可绕开这类收费问题，直接下载我们所需要的歌曲。

本章以 QQ 音乐为爬取对象，网站爬取范围是全站的歌曲信息，爬取方式是在歌手列表下获取每一位歌手的全部歌曲。由于爬取的数量较大，还会使用异步编程实现分布式爬虫开发，以提高爬虫效率。

整个爬虫项目按功能分为爬虫规则和数据入库，分别对应文件 music.py 和 music_db.py。

爬虫规则是在歌手列表（https://y.qq.com/portal/singer_list.html）中按照字母类别对歌手进行分类，遍历每个分类下的每位歌手页面，然后获取每位歌手页面下的全部歌曲信息。根据该设计方案列出遍历次数：

- 遍历每个歌手的歌曲页数。
- 遍历每个字母分类的每页歌手信息。
- 遍历每个字母分类的歌手总页数。
- 遍历 26 个字母分类和 1 个特殊符号的歌手列表。

在功能上至少需要实现 4 次遍历，但实际开发中往往比这个次数要多。统计遍历次数，主要能让开发者对项目开发有整体的设计逻辑。项目开发使用模块化设计思想，整个项目模块的划分如下：

- 歌曲下载。
- 歌手信息和歌曲信息。
- 字母分类下的歌手列表。
- 全站歌手列表。

18.2 歌 曲 下 载

下载歌曲前，先要找到歌曲的相关信息，才能够确定歌曲的下载链接。以 QQ 音乐中的某一首歌曲为例（https://y.qq.com/n/yqq/song/003OUlho2HcRHC.html），在 Chrome 浏览器访问网址，如图 18-1 所示。



图 18-1 歌曲信息页

在网页里单击“播放”按钮，浏览器会弹出一个新页面，在新页面里打开开发者工具并再次刷新网页，在 Netword 的 Media 选项卡可以找到歌曲播放文件，如图 18-2 所示。

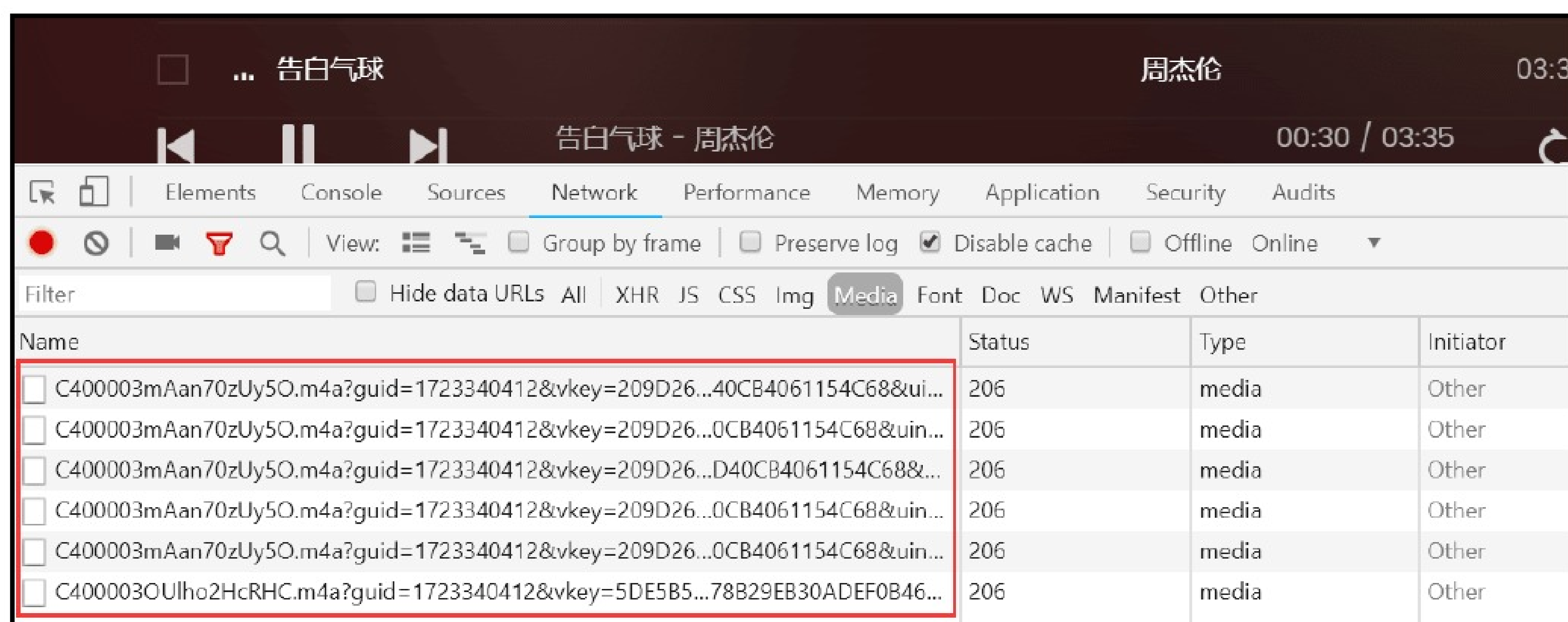


图 18-2 歌曲播放页

从图 18-2 中发现，Media 选项卡的歌曲文件有很多，但播放文件只有一个。分析 URL 内容发现，只有一个文件名与其他的文件名是不同的，如图中的 C400003OUlho2HcRHC.m4a，我们将其 URL 复制到浏览器的地址栏访问，发现歌曲可以播放，如图 18-3 所示。

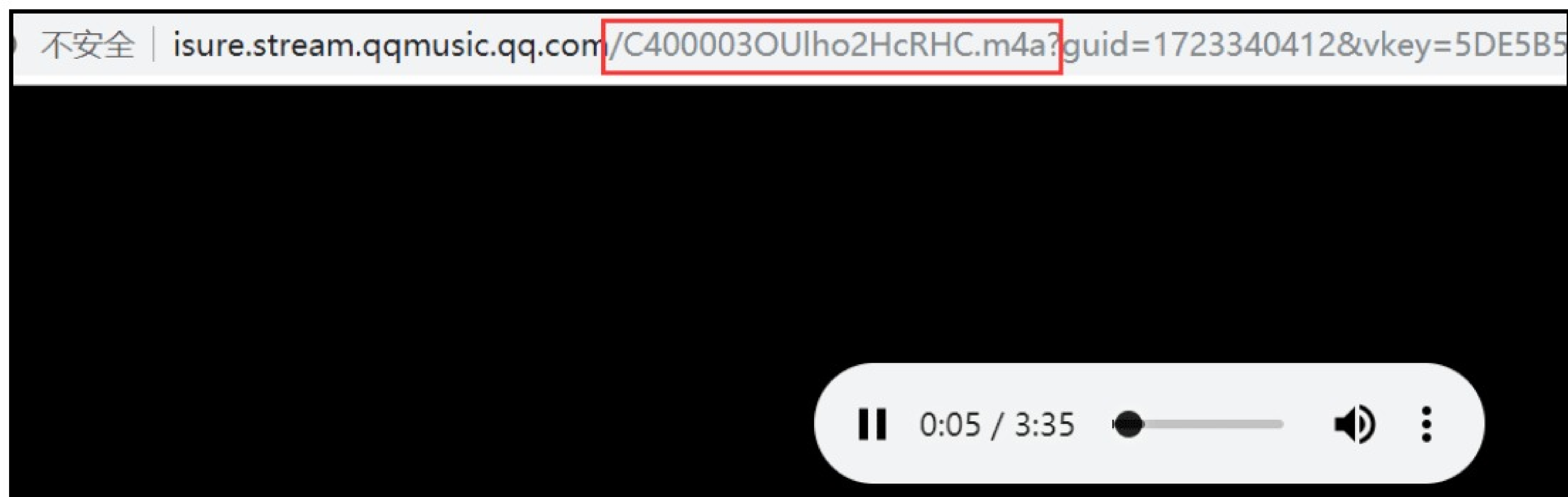


图 18-3　歌曲文件路径

从整个歌曲文件的 URL 结构分析可知，该 URL 是通过 GET 请求来访问歌曲文件，并且设有各种请求参数。若要实现歌曲下载，首先找到 URL 的请求参数。将某个请求参数进行复制，在 Network 的其他选项卡里，分别在每个请求信息的响应内容里查找这个请求参数，以参数 vkey 的值为例，在每个请求信息的响应内容使用“Ctrl+F”进行快速查找，最终在 JS 选项卡下找到该参数，如图 18-4 所示。

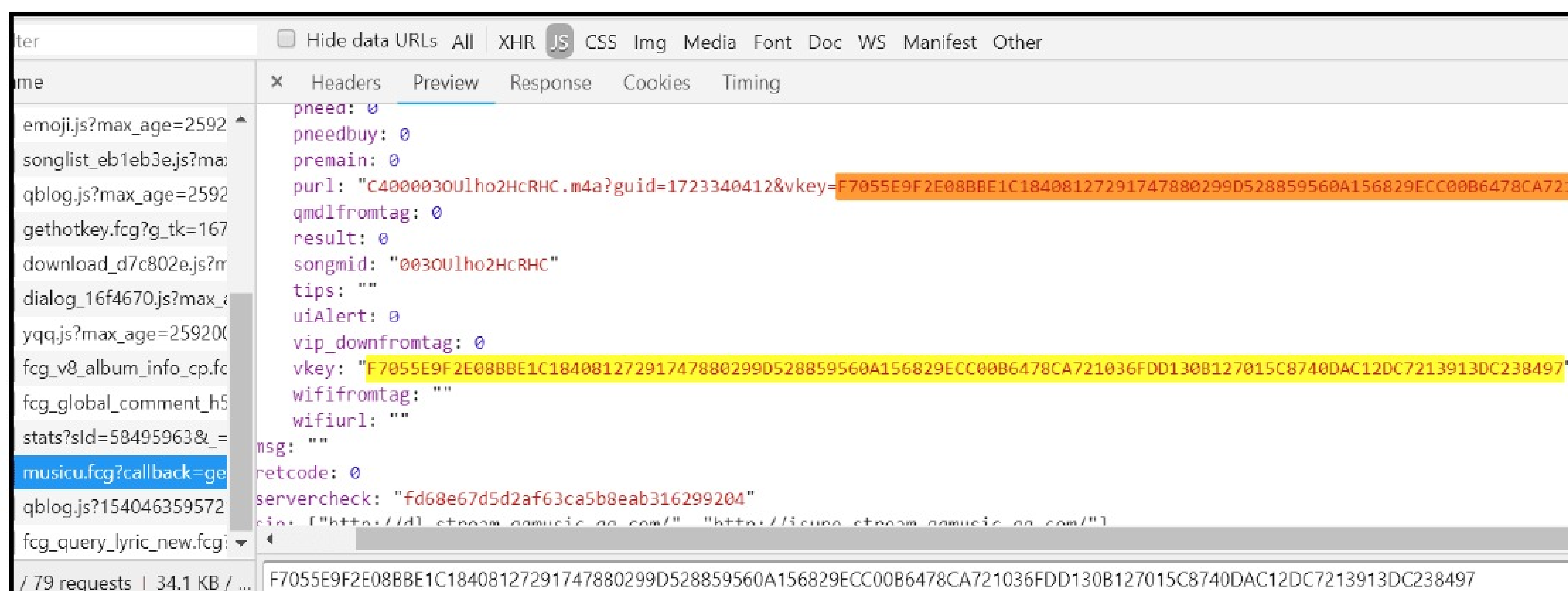


图 18-4　查找歌曲文件的请求参数

从图 18-4 上发现，purl 的值是歌曲文件路径的构成部分，只需再加上一个域名即可得到完整的歌曲文件路径。而对于域名的选择，QQ 音乐提供了 5 个可选域名，每个域名都可以获取歌曲文件，这是一种集群的管理方式。从图上的请求信息里可以找到具体的域名，如图 18-5 所示。



图 18-5　歌曲文件路径的域名

根据上述分析，目前能确定歌曲文件路径的请求参数是可以在 JS 选项卡的某个请求信息里获取。我们对这个请求信息的 URL 和请求参数进行分析，发现它的 URL 很长，并且设有复杂的请求参数。

根据请求参数的命名进行分析，并将 URL 放到浏览器的地址栏访问，然后逐一删除某些参数查看响应内容是否发生改变，若没改变，则该请求参数就可以直接去除。同时发现有两个请求参数的来源尚不明确，如图 18-6 所示。



图 18-6　请求信息

对于尚不明确的请求参数 guid 和 songmid，从命名方式来看，请求参数 songmid 是代表歌曲的唯一标识值，每首歌曲的 songmid 都是固定的。而参数 guid 却来自 Request Headers 的 Cookies，这是一种常见的反爬虫机制，同时发现每个请求信息都不是带有 Cookies。

将歌曲下载定为函数 download，并设置函数参数 guid、songmid 和 cookie_dict，分别代表请求参数 guid、songmid 和 Cookies，具体代码如下：

```
import requests, time
import math
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from music db import *
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
# 创建请求头和 requests 会话对象 session
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/
                    69.0.3497.100 Safari/537.36'
          }
```

```
session = requests.session()

# 下载歌曲
def download(guid, songmid,cookie_dict):
    # 参数 guid 来自 cookies 的 pgv pvid
    url = 'https://u.y.qq.com/cgi-bin/musicu.fcg?
            loginUin=0&hostUin=0&format=jsonp&inCharset=
            utf8&outCharset=utf-8&notice=0&platform=yqq&
            needNewCode=0&data=%7B%22req%22%3A%7B%22module%
            22%3A%22CDN.SrfCdnDispatchServer%22%2C%22method%
            22%3A%22GetCdnDispatch%22%2C%22param%22%3A%7B%22
            guid%22%3A%22'+guid+'%22%2C%22calltype%22%3A0%2C%
            22userip%22%3A%22%22%7D%7D%2C%22req 0%22%3A%7B%22
            module%22%3A%22vkey.GetVkeyServer%22%2C%22method
            %22%3A%22CgiGetVkey%22%2C%22param%22%3A%7B%22guid
            %22%3A%22'+guid+'%22%2C%22songmid%22%3A%5B%22'+
            songmid+'%22%5D%2C%22songtype%22%3A%5B0%5D%2C%22
            uin%22%3A%220%22%2C%22loginflag%22%3A1%2C%22platform%
            22%3A%2220%22%7D%7D%2C%22comm%22%3A%7B%22uin%22%3A0%
            2C%22format%22%3A%22json%22%2C%22ct%22%3A20%2C%22cv%
            22%3A0%7D%7D'
    r = session.get(url, headers=headers,cookies=cookie dict)
    purl = r.json()['req_0']['data']['midurlinfo'][0]['purl']
    # 下载歌曲
    if purl:
        url = 'http://isure.stream.qqmusic.qq.com/%s' %(purl)
        print(url)
        r = requests.get(url, headers=headers)
        f = open('song/' + songmid + '.m4a', 'wb')
        f.write(r.content)
        f.close()
        return True
    else:
        return False
```

上述代码导入了 Python 的模块，这是整个爬虫文件 music.py 所需的模块，而请求头 headers 和会话对象 session 是整个文件的全局变量。而函数 download 一共执行了两次 HTTP 请求，具体说明如下：

（1）通过函数参数 guid 和 songmid 来构建图 18-6 的 URL 地址；然后对该 URL 发送 GET 请求，并设有请求头和用户 Cookies 信息，这样可让 QQ 网站认为这次请求是合法的，并非是爬虫程序；最后将得到的响应内容并进行清洗，提取 purl 的值。

（2）判断 purl 的值是否为空，由于版权问题，QQ 音乐网站对某些歌曲没有播放权限，因此 purl 的值有可能为空。如果 purl 的值不为空，则通过 purl 的值来构建歌曲文件路径，然后向歌曲文件路径发送 GET 请求，将该请求的响应内容以字节的形式写入 m4a 文件，这样可完成歌曲的下载。

函数 download 的参数 guid 和 cookie_dict 是来自用户 Cookies 信息，而用户 Cookies 信息无法通过 Requests 模块获取，因此，我们需要借助 Selenium 实现。

通常情况下，使用 Selenium 访问网站就会自动生成相应的用户 Cookies 信息，但在 QQ 音乐网站来说，第一次访问是不会生成用户 Cookies 信息，直到第二次访问的时候才会生成用户 Cookies

信息。在第二次访问的时候还需要选择有用户 Cookies 信息的 URL，因为 QQ 音乐并不是任何 URL 都有用户 Cookies 信息。

根据上述的分析，我们将用户 Cookies 信息的获取定义为函数 getCookies，函数代码如下：

```
# 使用 Selenium 获取 Cookies
def getCookies():
    chrome_options = Options()
    # 设置浏览器参数
    # --headless 是不显示浏览器启动以及执行过程
    chrome_options.add_argument('--headless')
    driver = webdriver.Chrome(chrome_options=chrome_options)
    # 访问两个 URL，QQ 网站才能生成 Cookies
    driver.get('https://y.qq.com/')
    time.sleep(5)
    # 某个歌手的歌曲信息，用于获取 Cookies，因为不是全部请求地址都有 Cookies
    url = 'https://y.qq.com/n/yqq/singer/0025NhlN2yWrP4.html'
    driver.get(url)
    time.sleep(5)
    cookie = driver.get_cookies()
    driver.quit()
    # Cookies 格式化
    print(cookie)
    cookie_dict = {}
    for i in cookie:
        cookie_dict[i['name']] = i['value']
    return cookie_dict
```

函数 getCookies 在每次请求之间设置了 5 秒的等待时间，这是为了等待网站加载完成，否则网站尚未加载完成是无法读取用户 Cookies 信息的。现在将函数 getCookies 和 download 结合使用就能实现单首歌曲下载。在爬虫文件 music.py 编写以下代码并运行即可下载歌曲，在运行代码之前，记得在爬虫文件的路径下创建 song 文件夹。注意：函数参数 songmid 的获取会在下一节讲述。

```
if __name__=='__main__':
    cookie_dict = getCookies()
    guid = cookie_dict['pgv_pvid']
    songmid = '003OUlho2HcRHC'
    download(guid, songmid, cookie_dict)
```

18.3 歌手的歌曲信息

现在只要调用函数 download 并传入不同的参数 songmid 即可下载不同的歌曲，本节从歌手页面来获取不同歌曲的 songmid。以周杰伦（https://y.qq.com/n/yqq/singer/0025NhlN2yWrP4.html#tab=song）为例，打开歌手页面并在开发者工具查找歌曲信息，分别在 Doc、XHR 和 JS 里使用 Ctrl+F 快速查找某一首歌曲信息。最终在 JS 的某一条请求中找到歌曲信息，如图 18-7 所示。

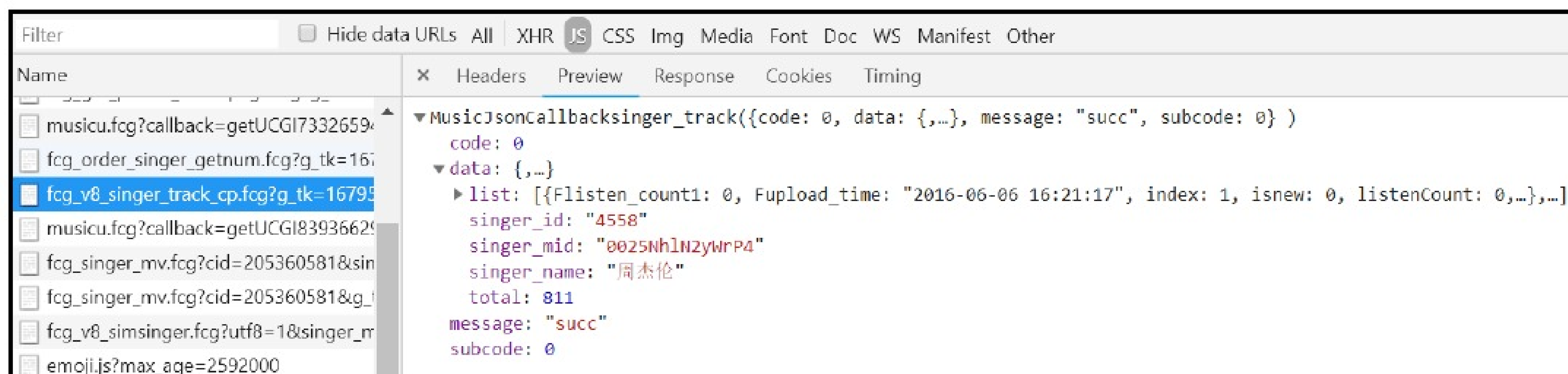


图 18-7　全部歌曲信息

从图 18-7 分析可得：

（1）total 是当前歌手的全部歌曲数目。

（2）list 是歌曲信息列表，每页共 30 首歌曲，对某首歌的信息进行分析，在信息中找到歌曲标识符、歌名、所属专辑和时长，分别对应 songmid、songname、albumname 和 interval，如图 18-8 所示。

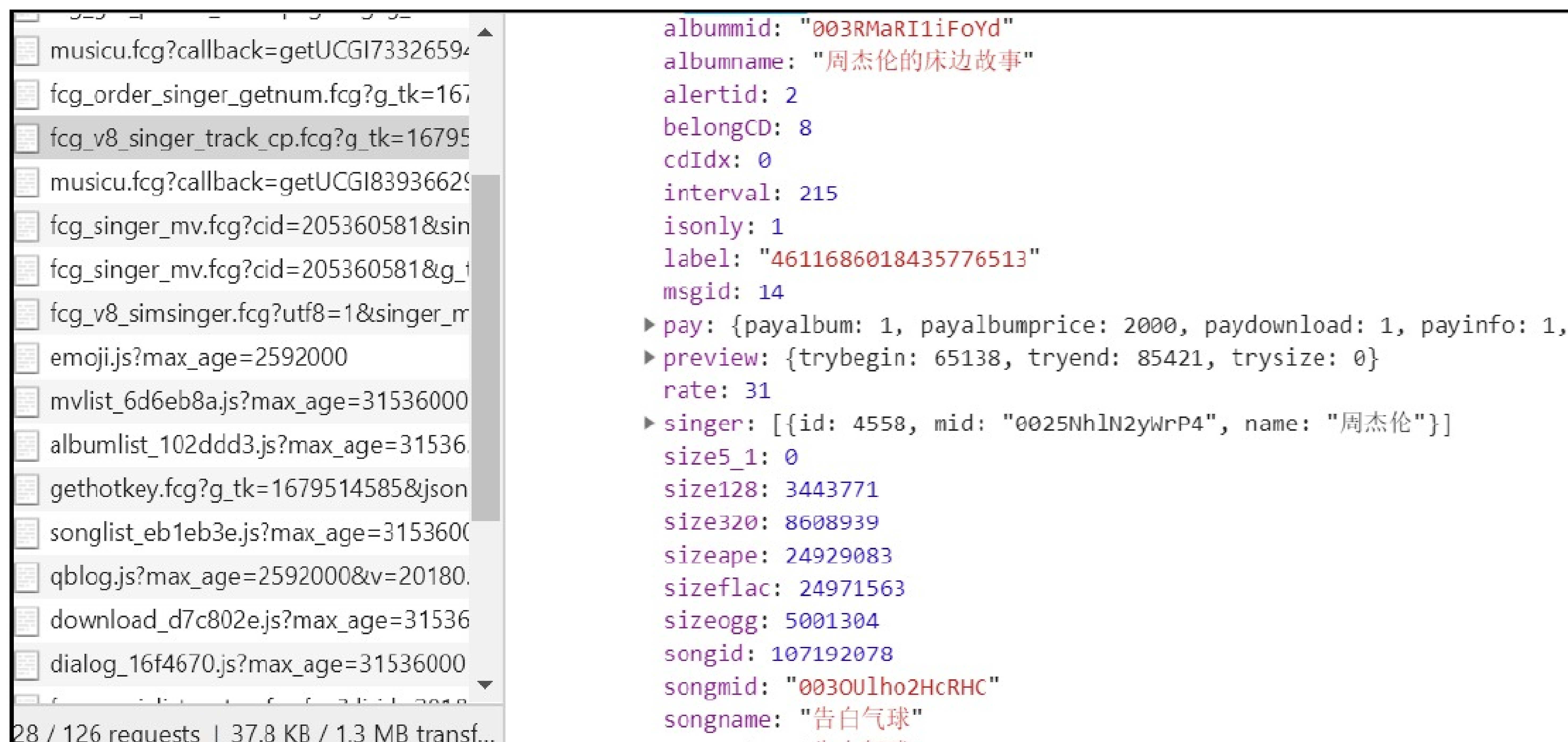


图 18-8　某个歌曲信息

对上述请求信息的 URL 进行分析，发现 URL 设有多个请求参数，将 URL 单独复制到浏览器上访问，并对每个请求参数进行删改，对比浏览器返回的响应内容是否和前面一致，最终 URL 的参数优化结果如图 18-9 所示。

对图 18-9 上的请求参数分析得知：

（1）singermid 是指歌手的 mid，其作用与 songmid 相同，用于标识歌手的唯一性。

（2）begin 代表歌曲页数，在网页上单击第二页的时候，会触发相同的 GET 请求，发现请求参数 begin 变为 30，说明页数不是按 1、2、3……计算的，而是按照（p-1）×30 的计算公式获取页数的。

（3）num 是每页歌曲数量的间隔数。



图 18-9　URL 的参数优化

综合分析，只要动态设置请求参数 singermid 和 begin 的值，就能获取不同歌手的全部歌曲信息。将其功能定义为函数 get_singer_songs，函数参数为 singermid 和 cookie_dict，函数代码如下：

```
# 获取歌手的全部歌曲
def get_singer_songs(singermid, cookie_dict):
    # 获取歌手姓名和歌曲总数
    url = 'https://c.y.qq.com/v8/fcg-bin/fcg_v8_singer_track_cp.fcg?
            loginUin=0&hostUin=0&singermid=%s&order=listen&begin=0&
            num=30&songstatus=1' % (singermid)
    r = session.get(url)
    # 获取歌手姓名
    song_singer = r.json()['data']['singer_name']
    # 获取歌曲总数
    songcount = r.json()['data']['total']
    # 根据歌曲总数计算总页数
    pagecount = math.ceil(int(songcount) / 30)
    # 循环页数，获取每一页歌曲信息
    for p in range(pagecount):
        url = 'https://c.y.qq.com/v8/fcg-bin/fcg_v8_singer_track_cp.fcg?
                loginUin=0&hostUin=0&singermid=%s&order=listen&begin=%s&
                num=30&songstatus=1' % (singermid, p * 30)
        r = session.get(url)
        # 得到每页的歌曲信息
        music_data = r.json()['data']['list']
        # songname-歌名，ablum-专辑，interval-时长，songmid 歌曲 id，用于下载音频文件
        # 将歌曲信息存放字典 song_dict，用于入库
        song_dict = {}
        for i in music_data:
            song_dict['song_name'] = i['musicData']['songname']
            song_dict['song_ablum'] = i['musicData']['albumname']
            song_dict['song_interval'] = i['musicData']['interval']
            song_dict['song_songmid'] = i['musicData']['songmid']
```

```
            song_dict['song_singer'] = song_singer
            # 下载歌曲
            guid = cookie_dict['pgv_pvid']
            info = download(guid,song_dict['song_songmid'],cookie_dict)
            # 入库处理，参数 song_dict
            if info:
                insert_data(song_dict)
            # song_dict 清空处理
            song_dict = {}
```

函数 get_singer_songs()用于爬取歌手的全部歌曲，代码说明如下：

（1）参数 singermid 代表歌手的唯一值，只需要传入不同歌手的 singermid，就能爬取不同歌手的全部歌曲。

（2）代码有两个相同变量 url，第一个是获取歌曲总数和歌手姓名，并通过歌曲总数计算页数；第二个是动态设置页数，获取当前歌手每一页的歌曲信息。

（3）下载歌曲调用了 18.2 节的函数 download，入库处理调用了入库函数 insert_data，该函数会在后续章节讲解。

18.4　分类歌手列表

现在已实现获取单个歌手的全部歌曲信息，只要在此功能的基础上遍历输入不同歌手的 singermid，就能获取不同歌手的歌曲信息。从开发者工具中对歌手列表（y.qq.com/portal/singer_list.html）的分析得知，歌手页数有 297 页，每页 80 位歌手，全站的歌手共有 23701 位，如图 18-10 所示。

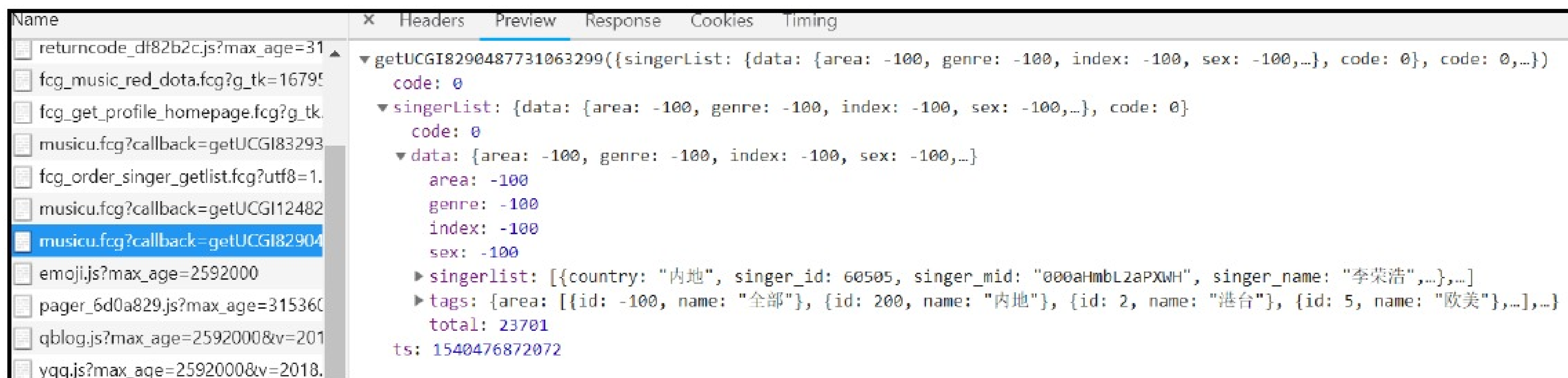


图 18-10　歌手信息

从图 18-10 上分析可知，singerlist 是每页 80 位歌手的信息列表，每条信息的 singer_mid 是歌手的 singermid。如果获取全部歌手的 singermid，需要循环 23701 次，根据项目设计，将循环次数按字母分类划分。

在歌手列表页上使用字母 A～Z 对歌手进行分类筛选，利用这个分类功能可以将全部歌手分为两层循环：第一层是循环每一个字母分类，第二层是循环每个分类下的总页数，拆分两层循环主要为异步编程提供切入点，具体实现方式会在后面讲解。

在网页上单击分类“A”，可在开发者工具的 JS 标签看到相应的请求信息，如图 18-11 所示。



图 18-11 分类歌手信息

对上述请求信息的 URL 进行分析，发现 URL 设有多个请求参数，由于该请求是一个 GET 请求，可以在浏览器上对 URL 的请求参数进行删减优化处理，最终 URL 的参数优化结果如图 18-12 所示。



图 18-12 URL 优化处理

我们分别单击不同的字母分类及切换不同页数，分析每个请求信息的请求参数，发现请求参数 index、sin 和 cur_page 有一定的变化规律：

（1）index 代表字母分类 A，从 1 开始，2 代表字母 B，以此类推。

（2）sin 是根据页数计算歌手数量，如第一页为 0，每页 80 位歌手，第二页为 80，第三页为 160，以此类推。

（3）cur_page 代表当前页数，从 1 开始，每页以 1 递增，如第二页为 2，以此类推。

综合上述分析，将分类歌手的 singermid 获取功能定义为函数 get_genre_singer，代码如下所示：

```
# 获取当前字母下全部歌手
def get_genre_singer(index, page_list, cookie_dict):
    for page in page_list:
        url = 'https://u.y.qq.com/cgi-bin/musicu.fcg?
                 loginUin=0&hostUin=0&format=jsonp&inCharset=utf8&
                 outCharset=utf8&notice=0&platform=yqq&needNewCode=0
                 &data=%7B%22comm%22%3A%7B%22ct%22%3A24%2C%22cv%22%3A
                 10000%7D%2C%22singerList%22%3A%7B%22module%22%3A%22
                 Music.SingerListServer%22%2C%22method%22%3A%22
                 get_singer_list%22%2C%22param%22%3A%7B%22area%22%3A-100%
                 2C%22sex%22%3A-100%2C%22genre%22%3A-100%2C%22index%22%3A'
```

```
                +str(index)+'%2C%22sin%22%3A'+str((page-1)*80)+'%2C%22
                cur page%22%3A'+str(page)+'%7D%7D%7D'
        r = session.get(url)
        # 循环每一个歌手
        for k in r.json()['singerList']['data']['singerlist']:
            singermid = k['singer mid']
            get_singer_songs(singermid, cookie_dict)
```

函数 get_genre_singer 用于爬取当前字母分类下全部歌手的歌曲信息，函数说明如下：

（1）index 代表字母的数字，如 1 代表字母 A，2 代表字母 B，以此类推。

（2）page_list 代表当前字母分类的总页数。

（3）cookie_dict 代表用户 Cookies 信息。

（4）外层 for 循环用于遍历当前分类的总页数。

（5）内层 for 循环用于遍历当前分类每页每位歌手的 singermid，并调用函数 get_singer_songs 获取每一位歌手的全部歌曲。

18.5　全站歌手列表

现在得到函数 get_genre_singer，只需传入不同的函数参数 index 和 page_list 就能获取不同分类的歌手列表。因此通过遍历 26 个字母和特殊符号#即可实现，将这个遍历定义为函数 get_all_singer，具体的代码如下：

```
def get all singer():
    # 获取字母 A-Z 全部歌手
    cookie dict = getCookies()
    for index in range(1, 28):
        # 获取每个字母分类下总歌手页数
        url = 'https://u.y.qq.com/cgi-bin/musicu.fcg?
                loginUin=0&hostUin=0&format=jsonp&inCharset=
                utf8&outCharset=utf-8&notice=0&platform=yqq&
                needNewCode=0&data=%7B%22comm%22%3A%7B%22ct%22
                %3A24%2C%22cv%22%3A10000%7D%2C%22singerList%22
                %3A%7B%22module%22%3A%22Music.SingerListServer
                %22%2C%22method%22%3A%22get_singer_list%22%2C%
                22param%22%3A%7B%22area%22%3A-100%2C%22sex%22%
                3A-100%2C%22genre%22%3A-100%2C%22index%22%3A'
                + str(index) + '%2C%22sin%22%3A0%2C%22cur page
                %22%3A1%7D%7D%7D'
        r = session.get(url, headers=headers)
        total = r.json()['singerList']['data']['total']
        pagecount = math.ceil(int(total) / 80)
        page list = [x for x in range(1, pagecount+1)]
        # 获取当前字母下全部歌手
        get genre singer(index, page list, cookie dict)
# 主程序运行
if   name   == '  main  ':
    get_all_singer()
```

函数 get_all_singer 的功能是构建参数 index、page_list 和 cookie_dict，具体说明如下：

（1）调用函数 getCookies，获取用户 Cookies 信息，将其作为参数 cookie_dict。
（2）执行 for 循环 27 次，循环从 1 开始至 27 结束，循环次数代表参数 index。
（3）每次循环代表每个字母，在当前循环下，通过发送请求来获取当前字母的歌手总页数。
（4）将总页数生成列表结构，作为参数 page_list。
（5）调用函数 get_genre_singer，从而实现全站歌曲下载。

上述函数 get_all_singer 也是整个项目程序的运行入口，程序运行执行函数的顺序如下：

- get_all_singer()：循环 26 个字母和特殊符号#，构建参数并调用函数 get_genre_singer。
- get_genre_singer(index, page_list, cookie_dict)：遍历当前分类总页数，获取每页每位歌手的歌曲信息。
- get_singer_songs(singermid, cookie_dict)：实现歌手的歌曲入库和下载。
- download(guid, songmid, cookie_dict)：下载歌曲。
- getCookies()：使用 Selenium 获取用户 Cookies 信息。
- insert_data(song_dict)：数据入库处理。

每个函数之间通过层层的调用来实现整个网站的歌曲下载和数据入库，每次函数调用都会传入不同的函数参数，使得函数之间存在一定的关联。

18.6 数据存储

在逻辑功能实现过程中发现数据入库使用的是函数 insert_data，该函数主要存放在 music_db.py 中，本节使用 SQLAlchemy 实现数据入库。

根据爬虫规则分析，入库的数据有歌名、所属专辑、时长、歌曲 mid（下载歌曲文件以歌曲 mid 命名）和歌手姓名。针对所爬取的数据及性质，数据库命名如表 18-1 所示。

表 18-1　song 数据表

字段	说明
song_id	主键
song_name	歌名
song_ablum	所属专辑
song_interval	时长
song_songmid	歌曲 mid
song_singer	歌手姓名

根据数据库的命名，SQLAlchemy 映射数据库及函数 insert_data 的代码如下：

```
from sqlalchemy import *
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative_base
# 连接数据库
```

```
    engine=create engine("mysql+pymysql://root:1234@localhost:
    3306/music db?charset=utf8")
    # 创建会话对象，用于数据表的操作
    DBSession = sessionmaker(bind=engine)
    SQLsession = DBSession()
    Base = declarative base()
    # 映射数据表
    class song(Base):
        # 表名
          tablename   ='song'
        # 字段、属性
        song id = Column(Integer, primary key=True)
        song name = Column(String(50))
        song_ablum = Column(String(50))
        song interval = Column(String(50))
        song songmid = Column(String(50))
        song_singer = Column(String(50))
    # 创建数据表
    Base.metadata.create all(engine)
    #定义函数 insert_data
    def insert data(song dict):
        # 连接数据库
        engine = create engine("mysql+pymysql://root:1234@localhost:
                                3306/music db?charset=utf8")
        # 创建会话对象，用于数据表的操作
        DBSession = sessionmaker(bind=engine)
        SQLsession = DBSession()
        data = song(
            song name = song dict['song name'],
            song ablum = song dict['song ablum'],
            song interval = song dict['song interval'],
            song songmid = song dict['song songmid'],
            song singer = song dict['song singer'],
        )
        SQLsession.add(data)
        SQLsession.commit()
```

函数 insert_data 主要对传递的参数 song_dict 进行入库处理，参数 song_dict 为字典格式。在函数里，重新创建新的数据库连接，这样做的目的是为异步编程而做准备的。

将上述代码存放在 music_db.py 文件中，在 music.py 中只需导入 music_db.py 的函数 insert_data 即可实现数据入库。

18.7　分布式爬虫

18.7.1　分布式概念

爬虫的爬取效率是实际生产中一个重要的考虑因素，时间就是金钱，更是一个企业能够生存下来的准则之一。为了提高爬虫的效率，本节为大家介绍一些异步编程的开发思想，简单地说，就

是利用多进程和多线程实现爬虫开发。

很多读者对 Python 的多线程有一定的误解，因为 Python 执行环境大部分依赖于 GIL，而 GIL 限制了多线程的功能。

需要明确的一点是，GIL 并不是 Python 的特性，它是在实现 Python 解析器（CPython）时所引入的一个概念。就好比 C++是一套语言（语法）标准，但是可以用不同的编译器来编译成可执行代码。有名的编译器有 GCC、INTEL C++，Visual C++等。Python 也一样，同样的代码可以通过 CPython、PyPy、Psyco 等不同的 Python 执行环境来执行。像其中的 JPython 就没有 GIL。然而因为 CPython 是大部分环境下默认的 Python 执行环境，所以在很多人的概念里 CPython 就是 Python，也就想当然地把 GIL 归结为 Python 语言的缺陷，其实 Python 完全可以不依赖于 GIL。

由于物理上的限制，各个 CPU 厂商在核心频率上的比赛已经被多核所取代。为了更有效地利用多核处理器的性能，就出现了多线程的编程方式，而随之带来的就是线程间数据一致性和状态同步的困难。

为了利用多核，Python 开始支持多线程，而解决多线程之间数据完整性和状态同步的最简单的方法就是加锁。于是有了 GIL 这把超级大锁，而当越来越多的代码库开发者接受了这种设定后，他们开始大量依赖这种特性（即默认 Python 内部对象是 thread-safe 的，无须在实现时考虑额外的内存锁和同步操作）。

Python 在设计之初就考虑到要在解释器的主循环中同时只有一个线程在执行，即在任意时刻，只有一个线程在解释器中运行。对 Python 虚拟机的访问由全局解释器锁（GIL）来控制，正是这个锁能保证同一时刻只有一个线程在运行。

在多线程环境中，Python 解释器按以下方式执行：

（1）设置 GIL。
（2）切换到一个线程去运行。
（3）运行：指定数量的字节码指令或者线程主动让出控制（可以调用 time.sleep(0)）。
（4）把线程设置为睡眠状态。
（5）解锁 GIL，再次重复以上所有步骤。

有人认为 Python 的多线程比较“鸡肋”，这种说法只是相对而言，Python 是仅有的支持多线程的解释型语言（比如 Perl 的多线程是残疾的，PHP 没有多线程）。相对自身而言，如果代码是 CPU 密集型，并且是线性执行，在这种情况下多线程就是“鸡肋”，效率可能还不如单线程；如果代码是 IO 密集型的，多线程可以明显提高效率，例如爬虫，在绝大多数时间都在等待服务器返回数据和频繁的 IO 数据读写。

18.7.2 并发库 concurrent.futures

Python 标准库为我们提供了 threading 和 multiprocessing 模块编写相应的多线程/多进程代码。从 Python 3.2 开始，标准库提供了 concurrent.futures 模块，它提供了 ThreadPoolExecutor 和 ProcessPoolExecutor 两个类，实现了对 threading 和 multiprocessing 更高级的抽象，对编写线程池/进程池提供了直接的支持。

下面通过简单的例子讲解如何使用 concurrent.futures，代码如下：

```
# 导入 concurrent.futures 模块
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
import datetime

# 线程的执行方法
def print_value(value):
    print('Thread' + str(value))

# 每个进程里面的线程
def myThread(value):
    Thread = ThreadPoolExecutor(max_workers=2)
    Thread.submit(print_value, datetime.datetime.now())
    Thread.submit(print_value, datetime.datetime.now())

# 创建两个进程，每个进程执行 myThread 方法，myThread 主要将每个进程通过线程执行
# 如果不填写 max_workers=2，根据计算机的 CPU 核数创建进程，如果四核就创建 4 个进程
def myProcess():
    pool = ProcessPoolExecutor(max_workers=2)
    pool.submit(myThread, datetime.datetime.now())
    pool.submit(myThread, datetime.datetime.now())

if __name__ == '__main__':
    myProcess()
```

在上述代码中，创建了进程 ProcessPoolExecutor 和线程 ThreadPoolExecutor，其中在每个进程中又创建了两个线程。下面简单讲述一下 concurrent.futures 属性和方法。

（1）Executor：Executor 是一个抽象类，它不能被直接使用。为具体的异步执行定义了基本的方法：ThreadPoolExecutor 和 ProcessPoolExecutor 继承了 Executor，分别被用来创建线程池和进程池的代码。

（2）创建进程和线程之后，Executor 提供了 submit()和 map()方法对其操作。submit()和 map()最大的区别是参数类型，map()的参数必须是列表、元组和迭代器的数据类型。

（3）Future：可以理解为一个在未来完成的操作，这是异步编程的基础。通常情况下，执行 IO 操作和访问 URL 时，在等待结果返回之前会产生阻塞，CPU 不能做其他事情，而 Future 的引入帮助我们在等待的这段时间可以完成其他的操作。

18.7.3　分布式策略

我们知道，爬取全站歌曲信息是按照字母 A～Z 和符号#依次爬取，这是在单进程单线程的情况下运行。如果将这 27 次循环分为 27 个进程同时执行，每个进程只需执行对应的字母分类，假设执行一个字母分类的爬取时间相同，那么多进程并发的效率是单进程的 26 倍。

除了运用多进程之外，项目代码大部分是 IO 密集型，那么在每个进程下使用多线程也可以提高每个进程的运行效率。歌手列表页是通过两层循环实现，第一层是循环每个分类字母，现将每个分类字母作为一个单独进程处理；第二层是循环每个分类的歌手总页数，可将这个循环使用多线程处理。假设每个进程使用 10 条线程（线程数可自行设定，具体看实际需求），那么每个进程的效率也相对提高 10 倍。

分布式策略考虑的因素有网站服务器负载量、网速快慢、硬件配置和数据库最大连接量。举个例子，爬取某个网站 1000 万数据，从数据量分析，当然进程和线程越多，爬取的速度越快。但往往忽略了网站服务器的并发量，假设设定 10 个进程，每个进程 200 条线程，每秒并发量为 200×10=2000，若网站服务器并发量远远低于该并发量，在发送请求到网站的时候，就会出现卡死的情况，导致请求超时（即使对超时做了相应处理），无形之中增加了等待时间。除此之外，进程和线程越多，对运行爬虫程序的系统的压力越大，若涉及数据入库，还要考虑并发数是否超出数据库的最大连接数的情况。

根据上述分布式策略，在 music_db.py 中添加如下代码：

```
# 多线程
def myThread(index, cookie_dict):
    # 每个字母分类的歌手列表页数
    url = 'https://u.y.qq.com/cgi-bin/musicu.fcg?
            loginUin=0&hostUin=0&format=jsonp&inCharset=
            utf8&outCharset=utf-8&notice=0&platform=yqq&
            needNewCode=0&data=%7B%22comm%22%3A%7B%22ct%2
            2%3A24%2C%22cv%22%3A10000%7D%2C%22singerList%
            22%3A%7B%22module%22%3A%22Music.SingerListServer
            %22%2C%22method%22%3A%22get_singer_list%22%2C%22
            param%22%3A%7B%22area%22%3A-100%2C%22sex%22%3A-1
            00%2C%22genre%22%3A-100%2C%22index%22%3A' +
            str(index) + '%2C%22sin%22%3A0%2C%22cur_page%22%
            3A1%7D%7D%7D'
    r = session.get(url, headers=headers)
    total = r.json()['singerList']['data']['total']
    pagecount = math.ceil(int(total) / 80)
    page_list = [x for x in range(1, pagecount+1)]
    thread_number = 10
    # 将每个分类总页数平均分给线程数
    list_interval=math.ceil(len(page_list)/thread_number)
    # 设置线程对象
    Thread = ThreadPoolExecutor(max_workers=thread_number)
    for i in range(thread_number):
        # 计算每条线程应执行的页数
        start_num = list_interval * i
        if list_interval * (i + 1) <= len(page_list):
            end_num = list_interval * (i + 1)
        else:
            end_num = len(page_list)
        # 每个线程各自执行不同的歌手列表页数
        Thread.submit(get_genre_singer, index,
                      page_list[start_num: end_num],cookie_dict)

# 多进程
def myProcess():
    with ProcessPoolExecutor(max_workers=27) as executor:
        cookie_dict = getCookies()
        for index in range(1, 28):
            # 创建 27 个进程，分别执行 A-Z 分类和特殊符号#
            executor.submit(myThread, index, cookie_dict)
```

```
    # 主程序运行
    if __name__ == '__main__':
        myProcess()
```

代码中定义了函数 myProcess 和 myThread，分别实现多进程和多线程。

（1）多进程函数 myProcess：主要是循环字母 A～Z 和符号#，将每个字母独立创建一个进程，每个进程执行函数是 myThread，参数是当前的分类字母和用户 Cookies 信息。

（2）多线程函数 myThread：首先传入参数 index 来获取当前分类的歌手总页数，然后得到的总页数和设定的线程数计算每条线程应执行的页数，最后遍历设定的线程数，让每条线程执行相应的页数。例如总页数 100 页，10 条线程，每条线程应执行 10 页，第一条线程执行 0～10 页，第二条线程执行 10～20 页，以此类推。线程调用的函数是 get_genre_singer。

在实现分布式爬虫的时候，必须注意的是：

（1）全局变量不能放在 if __name__ =='__main__'中，因为使用多进程的时候，新开的进程不会在此获取数据。

（2）使用 SQLalchemy 入库最好重新创建一个数据库连接，如果多个线程和进程共同使用一个连接，就会出现异常。

（3）分布式策略最好在程序代码的最外层实现。例如在项目中，函数 get_singer_songs 里有两个 for 循环，不建议在此使用分布式处理，在代码底层实现分布式不是不可行，只是代码变动太大，而且考虑的因素较多，代码维护相对较难。

18.8　本章小结

本章以 QQ 音乐为爬取对象，爬取范围是全站的歌曲信息，爬取方式在歌手列表获取每一位歌手的全部歌曲。如果爬取的数量较大，就使用异步编程实现分布式爬虫开发，可提高爬虫效率。读者应重点掌握以下内容：

1. 项目实现的功能

（1）download(guid, songmid, cookie_dict)：歌曲下载，这是爬虫最底层的功能。

（2）get_singer_songs(singermid, cookie_dict)：将歌手的歌曲信息入库和歌曲下载。

（3）get_genre_singer(index, page_list, cookie_dict)：获取字母分类的全部歌手和歌曲信息。

（4）get_all_singer()：循环 26 个字母和特殊符号#，构建参数并调用函数 get_genre_singer。

（5）insert_data(song_dict)：将爬取的歌手和歌曲信息入库处理。

（6）myProcess()：每个字母分类创建一个单独进程运行。

（7）myThread(index, cookie_dict)：每个进程使用多线程爬取数据。

2. 分布式策略考虑的因素

分布式策略考虑的因素有网站服务器负载量、网速快慢、硬件配置和数据库最大连接量。举个例子，爬取某个网站 1000 万数据，从数据量分析，当然进程和线程越多，爬取的速度越快。但往往忽略了网站服务器的并发量，假设设定 10 个进程，每个进程 200 条线程，每秒并发量为

200×10=2000，若网站服务器并发量远远低于该并发量，在发送请求到网站的时候，就会出现卡死的情况，导致请求超时（即使对超时做了相应处理），无形之中增加了等待时间。除此之外，进程和线程越多，对运行爬虫程序的系统的压力越大，若涉及数据入库，还要考虑并发数是否超出数据库的最大连接数的情况。

3. 实现分布式爬虫的注意事项

（1）全局变量不能放在 if __name__ == '__main__'中，因为使用多进程的时候，新开的进程不会在此获取数据。

（2）使用 SQLalchemy 入库最好重新创建一个数据库连接，如果多个线程和进程共同使用一个连接，就会出现异常。

分布式策略最好在程序代码的最外层实现。例如在项目中，函数 get_singer_songs 里有两个 for 循环，不建议在此使用分布式处理，在代码底层实现分布式不是不可行，只是代码变动太大，而且考虑的因素较多，代码维护相对较难。

提　示
本章项目仅用于爬虫教学，并无违反版权法规之意，请读者注意自觉树立版权保护意识。

第 19 章

实战：12306 抢票爬虫

19.1 项 目 分 析

12306 抢票是爬虫开发中非常经典的一个项目。官方为了打击黄牛囤票，网站不断地更新升级，各类抢票软件也不断地修正更改，两者周而复始，印证了一句话“程序员都是在互相伤害”。

这种抢票类爬虫的开发思路与用户在浏览器的购票操作一致，只不过是编写代码来完成购票流程，可以理解为通过程序来模拟用户在浏览器上购买车票。

在本项目中，按照购买火车票的流程：用户登录→查询车票信息→选择班次→填写乘车人员信息→提交并生成订单，制定爬虫功能开发顺序，即：

（1）验证码验证。
（2）用户登录与验证。
（3）查询车票。
（4）预订车票。
（5）提交订单。
（6）生成订单。

19.2 验证码验证

在 12306 网站购买火车票的时候，首先需要用户登录，除了需要输入账号、密码之外，还设有图片验证码，验证码的验证方式是根据图片中的问题选择正确的答案，当验证码和账号信息正确时，才能登录成功。

用爬虫实现登录功能，首先需分析网站的登录事件所触发的请求信息。在 Chrome 浏览器中访问 12306 的用户登录界面（https://kyfw.12306.cn/otn/login/init）。该登录界面除了账号、密码输入

框之外，还有一个图片验证码，图片验证码是由一个问题描述和 8 组图片组成的。打开开发者工具，在登录界面输入账号、密码并选择错误的验证码答案，最后单击登录按钮，可以看到有两个请求信息，如图 19-1 所示。

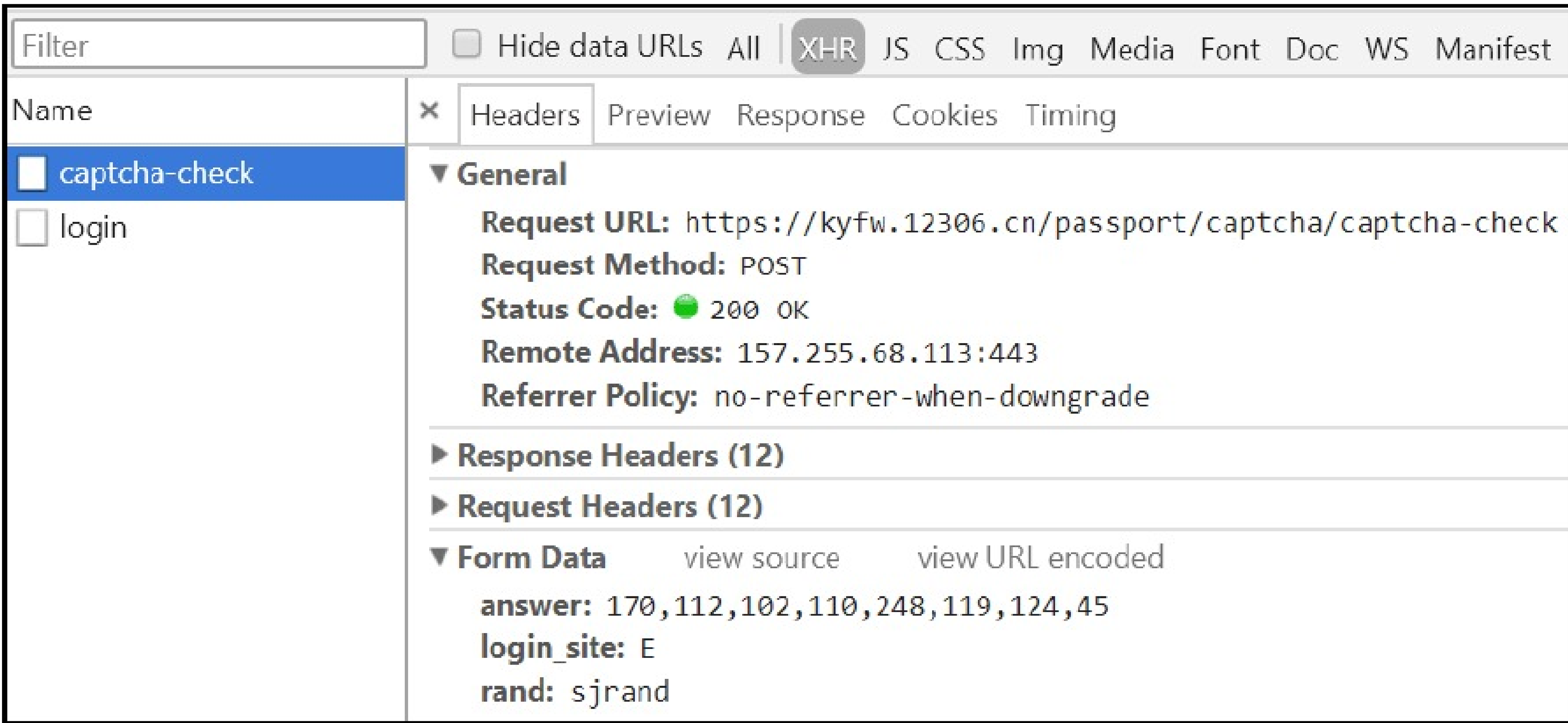


图 19-1 验证码验证

从图 19-1 可以看到，当输入正确的账号、密码并选择错误的验证码答案登录后，会触发两个 POST 请求，其中第一个请求链接是 https://kyfw.12306.cn/passport/captcha/captcha-check，从 URL 组成和响应内容分析，该请求信息的作用是验证用户选择的答案与验证码答案是否符合，如图 19-2 所示。

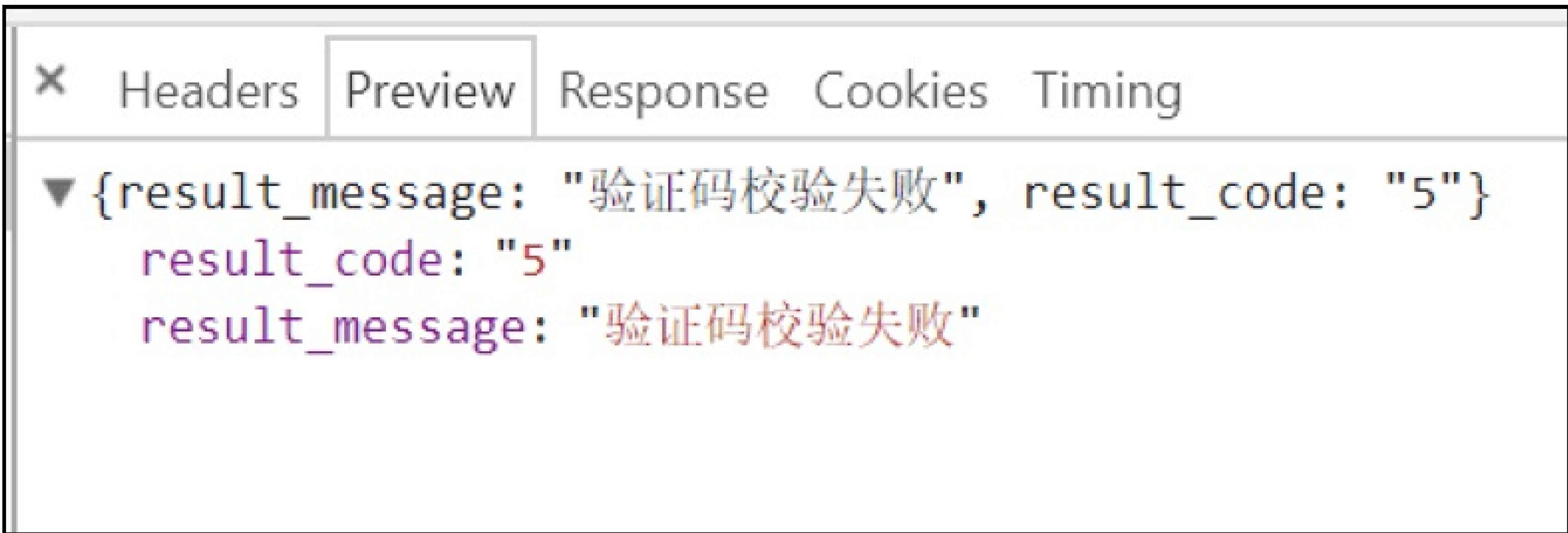


图 19-2 验证码校验结果

从图 19-1 可知，验证码校验请求有三个参数，分别是 answer、login_site 和 rand。单从一次请求信息是无法找出请求参数的变化规律的，不妨重复多次上述操作，观察请求参数的变化来判断数据规律。通过多次操作（输入正确的账号、密码，并选择错误而不重复的验证码答案），发现参数 login_site 和 rand 的参数值是固定不变的，而参数 answer 会根据每次选择的答案的不同而不断地变化。

为了进一步找出参数 answer 的变化规律，尝试以下方法：

（1）第一次只选择第一组图片，参数 answer 的值为 40,40。

（2）第二次只选择第二组图片，参数 answer 的值为 114,35。

（3）第三次只选择第三组图片，参数 answer 的值为 192,39。

（4）以此类推，第四、第五、第六、第七和第八组图片分别对应 257,36、42,115、119,107、185,124 和 272,117。也就是说，每组图片对应一组数字。

通过多次试验发现，同一组图片，根据单击位置的不同，参数 answer 的值随之变化，如第一组图片，单击位置分别在左上方和左下方，其参数 answer 的值有所不同。根据这样的变化，每组数字应该代表一个坐标位置，每组图片代表一定的区域范围，只要坐标位置在图片的区域范围内，这组数据就代表这张图片。这种验证码称之为坐标验证码，这种坐标系的验证码属于图片验证码的一种类型，对于这种验证码目前还没有很好的解决方案，只能通过人为输入正确的坐标位置来完成验证。

综合分析，可以确定验证码里面的 8 组图片的坐标位置（每组图片的坐标位置不是唯一的，只要坐标位置在图片的区域内即可），验证码校验代码如下：

```
import requests
# 坐标参考：40,40,114,35,192,39,257,36,42,115,119,107,185,124,272,117
code list={
    '1': '40,40,',
    '2': '114,35,',
    '3': '192,39,',
    '4': '257,36,',
    '5': '42,115,',
    '6': '119,107,',
    '7': '185,124,',
    '8': '272,117'
}
# 创建会话
session = requests.session()
# 请求头
headers = { 'User-Agent':
            'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 '
            '(KHTML, like Gecko) Chrome/63.0.3218.0 Safari/537.36',
            'Referer':
            'https://kyfw.12306.cn/otn/login/init'}
# 验证码图片的 URL
url = 'https://kyfw.12306.cn/passport/captcha/captcha-image?
login site=E&module=login&rand=sjrand'
# 忽略证书验证
r = session.get(url, headers=headers, verify=False)
#下载验证码图片
F = open('code.png','wb')
f.write(r.content)
f.close()
# 输入验证码图片位置，每组图片用英文逗号隔开
code = input("请输入验证码：")
get code = ''
for i in code.split(','):
    # 根据输入每组图片的组号获取对应的坐标位置
    get code += code list[i]
# 验证码校验
data = {
    'answer': get code,
    'login site': 'E',
    'rand': 'sjrand'
}
url = 'https://kyfw.12306.cn/passport/captcha/captcha-check'
```

```
    r = session.post(url, data=data)
    print(r.text)
```

整段代码实现了两次请求，第一次请求是下载验证码图片，第二次请求是对验证码进行校验。代码细节如下。

- code_list：使用字典数据格式，主要用于验证码识别，用户可直接输入每组图片的组号获取对应的坐标位置。
- session = requests.session()：创建一个持久化会话对象，确保每一次请求在同一个会话中。
- verify=False：忽略证书验证。如果没有安装 12306 网站的根证书，爬取过程中会提示连接不安全而导致无法访问，一般忽略证书验证即可解决。
- 验证码识别：根据验证码的问题找出图片所在组号的位置即可。图片位置顺序是从上到下再从左到右，组号从 1 到 8 依次排序。如果有多个组号，组号之间就用英文的逗号隔开。

验证码验证：将输入的字符串以逗号分割后，根据图片位置找到对应的图片坐标，最后将坐标拼接起来就得到参数 answer。

运行上述代码，结果如图 19-3 所示。



图 19-3　验证结果

19.3　用户登录与验证

完成验证码验证后，下一步是实现用户登录功能。从图 19-1 可知，单击登录按钮会触发两个 POST 请求，其中第二个是用户登录请求，对该请求进行分析，如图 19-4 所示。

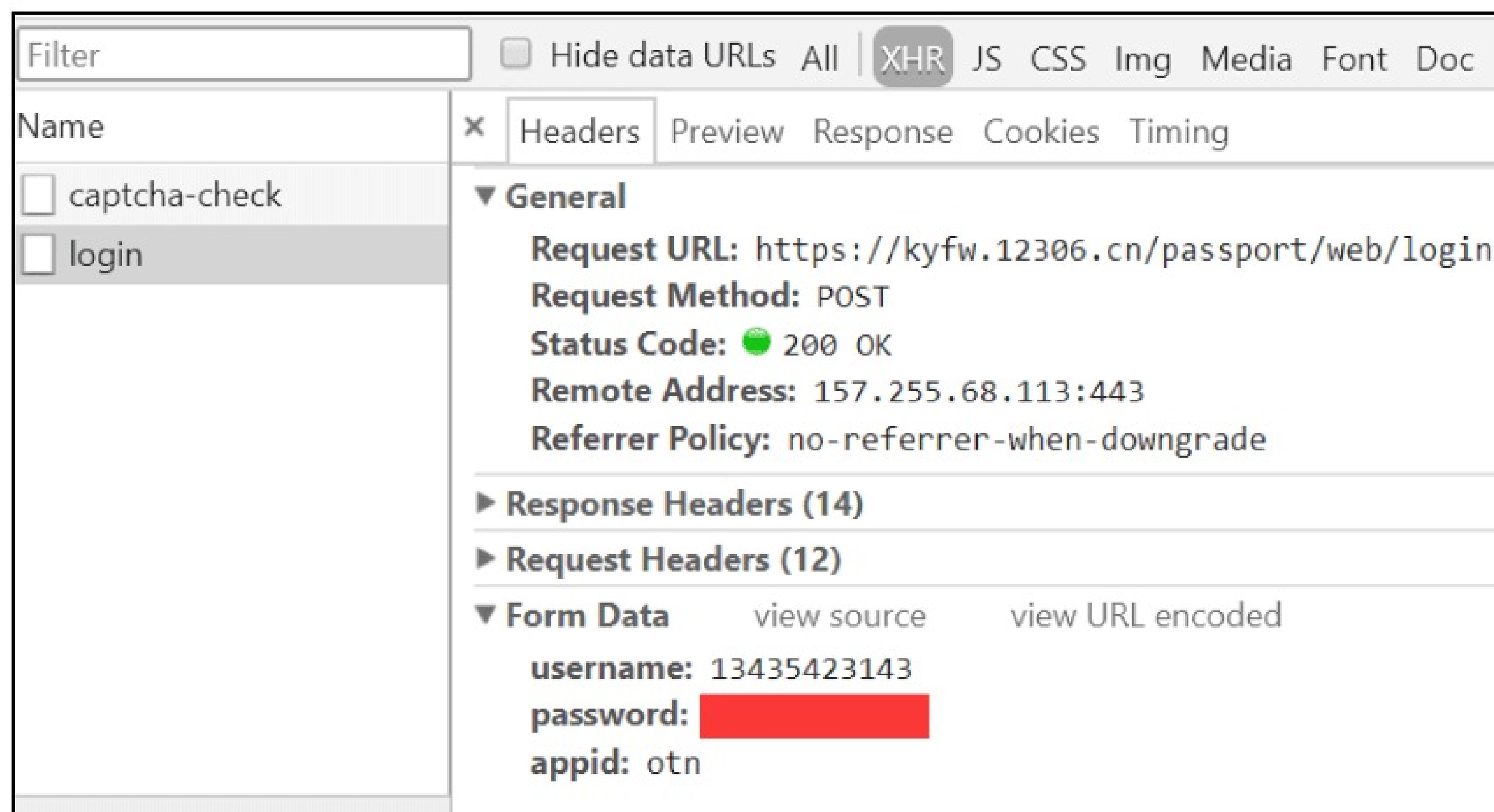


图 19-4　登录请求

登录请求的参数有 username、password 和 appid，而且 username 和 password 没有经过加密处理，参数 appid 是固定不变的。那么，用户登录的代码如下：

```
url ='https://kyfw.12306.cn/passport/web/login'
data = {
    'username': '13435423143',
    'password': 'XXXXXX',
    'appid': 'otn'
}
r = session.post(url, data=data)
print(r.text)
```

由于用户登录的代码无法单独运行，因此我们将验证码验证和用户代码整合在一起，只要将用户登录的代码添加到验证码的代码下面即可，运行结果如图 19-5 所示。

```
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request
  InsecureRequestWarning)
请输入验证码：5,6
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request
  InsecureRequestWarning)
{"result_message":"验证码校验成功","result_code":"4"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request
  InsecureRequestWarning)
{"result_message":"登录成功","result_code":0,"uamtk":"0njyEiC1OmhGTRb5W3vPv9E3n45XPhAk-5Msuw4mEnwsd1110"}
```

图 19-5　用户登录信息

完成用户登录后，接着回到登录界面，当输入正确的账号、密码和验证码之后，单击登录按钮，触发两个请求之后，发现网页会自动跳转，在网页跳转时，在开发者工具捕捉到两个新的 POST 请求，但这两个请求只在一瞬间出现，等网页跳转完成之后，请求信息就会被清理掉。

这是因为 HTTP 的 302 网页跳转之后，Chrome 浏览器将之前捕捉到的请求清空，然后重新捕捉新页面的请求，遇到这种情况，只能在网页跳转的期间内单击开发者工具，然后按 Ctrl+E 停止 Network 对请求的捕捉，这样就能保留之前的请求信息。除此之外，读者还可以使用 Fiddler 对网站抓包。

回到本项目中，在网页跳转之前，Chrome 浏览器捕捉的请求信息如图 19-6 和图 19-7 所示。

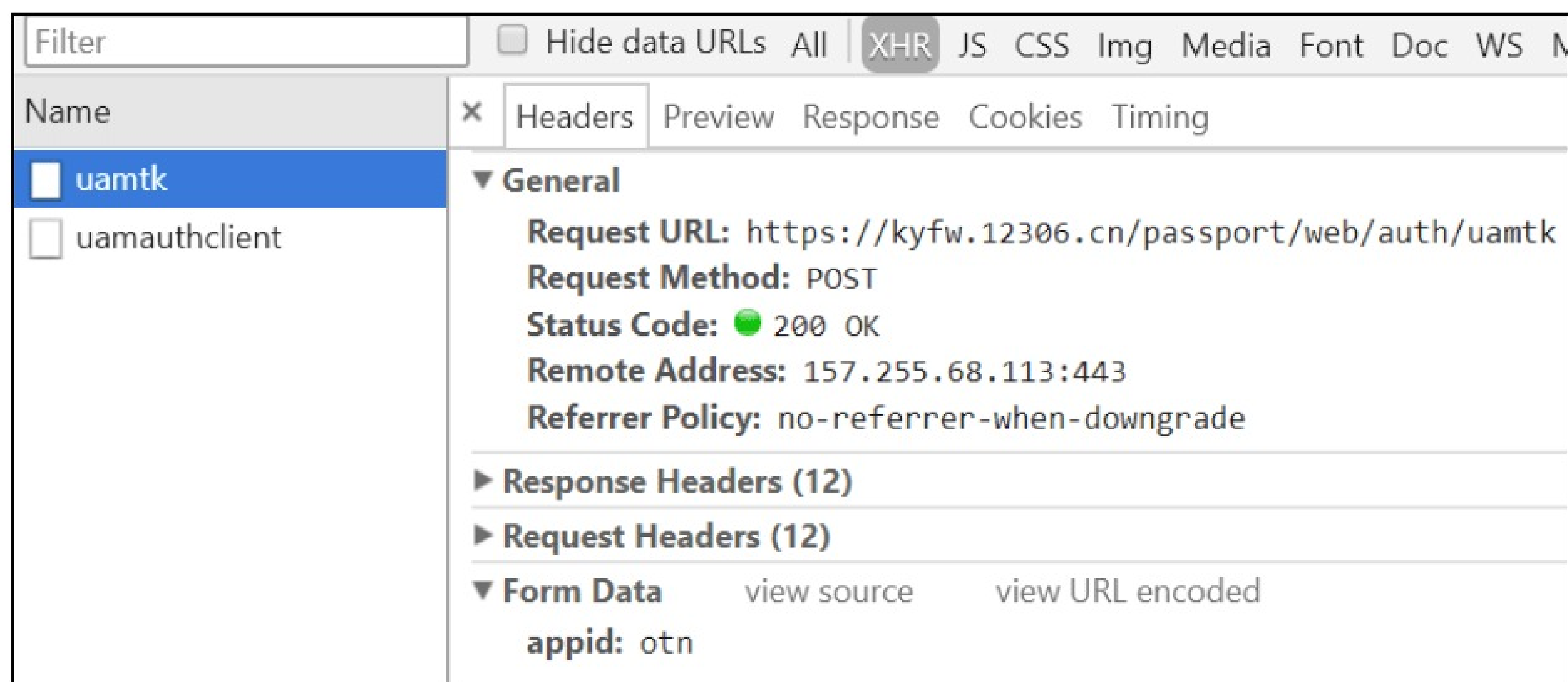


图 19-6　用户登录验证一

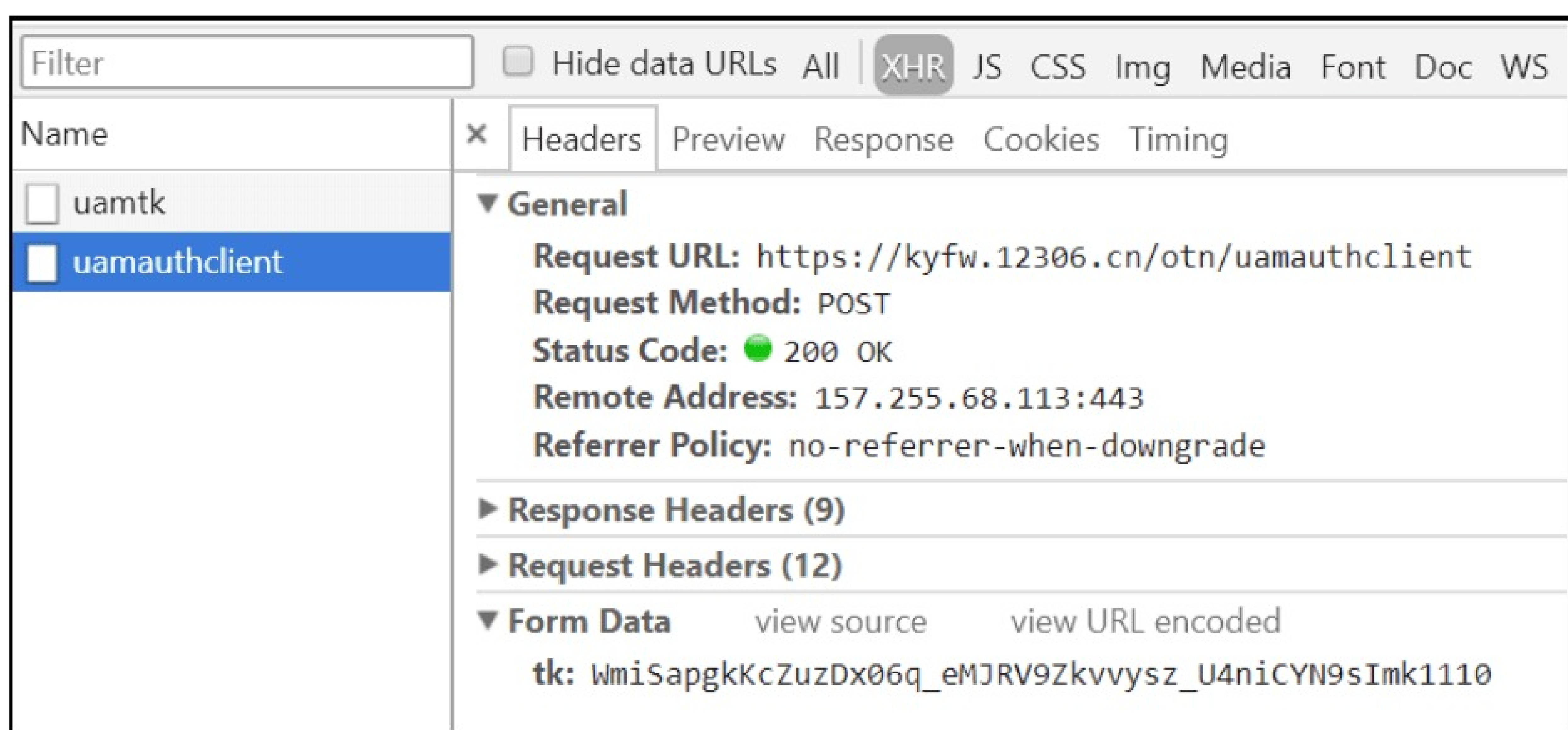


图 19-7　用户登录验证二

从图 19-6 和图 19-7 可知，网页跳转时，发生了两次 POST 请求，而且两者只有一个请求参数，图 19-6 的请求参数是固定值，而图 19-7 的请求参数是变化值。

现在无法确定图 19-7 请求参数的由来。一般来说，请求参数主要的来源如下：

（1）Doc 标签的 HTML 内容，可以复制参数值的内容，然后在 HTML 中快速查找参数是否存在。

（2）XHR 标签的请求信息，参数可能由其他的请求信息生成。

（3）JS 标签的请求信息，需要对 JavaScript 代码进行解读，参数有可能由 JavaScript 生成。

（4）特殊数据，如随机数和时间戳。随机数大多数都是以小数为主的，大多数随机数可以视为固定不变的参数；时间戳通常以 150XXXXX 开头，长度一般为 9～16 位不等。

根据上述查找方法结合实际分析，当前只有两个 POST 请求，第二个请求信息的请求参数很可能来自于第一个请求信息的响应内容。

在上述已完成的代码中添加以下代码，实现图 19-6 的请求：

```
url = 'https://kyfw.12306.cn/passport/web/auth/uamtk'
```

```
    data = {
        'appid': 'otn'
    }
    r = session.post(url, data=data)
    print(r.text)
```

运行代码，结果如图 19-8 所示。

```
  InsecureRequestWarning)
请输入验证码：4,5
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made.
  InsecureRequestWarning)
{"result_message":"验证码校验成功","result_code":"4"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made.
  InsecureRequestWarning)
{"result_message":"登录成功","result_code":0,"uamtk":"9SHmBE6TX_imlaKVaUD_TwqxT3WLyRDR-r05nMwPiPwlp1110"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made.
  InsecureRequestWarning)
{"result_message":"验证通过","result_code":0,"apptk":null,"newapptk":"tg_aFvSVVZAtN8srMpIXMM4gHIRb-BGL-Euop4hHVcket1110"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made.
```

图 19-8　用户登录验证结果一

从最后的输出结果分析，newapptk 的数据格式和图 19-7 的请求参数比较符合。尝试使用 newapptk 的数据作为图 19-7 的请求参数，在上述代码中添加以下代码：

```
    # newapptk 是图 15-6 请求之后的响应结果
    newapptk = r.json()['newapptk']
    url = 'https://kyfw.12306.cn/otn/uamauthclient'
    data = {
        'tk': newapptk
    }
    r = session.post(url, data=data)
    print(r.text)
```

运行结果如图 19-9 所示。

```
请输入验证码：4,5
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding
  InsecureRequestWarning)
{"result_message":"验证码校验成功","result_code":"4"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding
  InsecureRequestWarning)
{"result_message":"登录成功","result_code":0,"uamtk":"9SHmBE6TX_imlaKVaUD_TwqxT3WLyRDR-r05nMwPiPwlp1110"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding
  InsecureRequestWarning)
{"result_message":"验证通过","result_code":0,"apptk":null,"newapptk":"tg_aFvSVVZAtN8srMpIXMM4gHIRb-BGL-Euop4hHVcket1110"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding
  InsecureRequestWarning)
{"apptk":"tg_aFvSVVZAtN8srMpIXMM4gHIRb-BGL-Euop4hHVcket1110","result_code":0,"result_message":"验证通过","username":"黄永祥"}
```

图 19-9　用户登录验证结果二

根据图 19-9 的运行结果得知，第二次 POST 的请求参数（见图 19-7）来自于第一次 POST 请求（见图 19-6）的响应内容，也就说这两个请求是紧密联系的，共同完成用户登录验证功能。

用户登录网站由三部分功能组成：验证码验证→用户登录→用户验证。对这三部分功能进行优化和整理，代码如下：

```
import requests
def login(username, password):
    # 坐标参考：40,40,114,35,192,39,257,36,42,115,119,107,185,124,272,117
    code_list = {
        '1': '40,40,',
        '2': '114,35,',
        '3': '192,39,',
        '4': '257,36,',
        '5': '42,115,',
        '6': '119,107,',
        '7': '185,124,',
        '8': '272,117'
    }
    # 请求头
    headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; WOW64)
                             AppleWebKit/537.36 (KHTML, like Gecko)
                              Chrome/63.0.3218.0 Safari/537.36',
              'Referer':'https://kyfw.12306.cn/otn/login/init'}

    url = 'https://kyfw.12306.cn/passport/captcha/captcha-image?
           login_site=E&module=login&rand=sjrand'
    # 忽略证书验证
    r = session.get(url, headers=headers, verify=False)
    # 下载验证码图片
    f = open('code.png','wb')
    f.write(r.content)
    f.close()
    # 输入验证码图片位置，多个验证码用英文逗号分开
    code=input("请输入验证码：")
    get_code = ''
    for i in code.split(','):
        # 根据输入每组图片的组号获取对应的坐标位置
        get_code += code_list[i]
    # 验证码校验
    data={
        'answer':get_code,
        'login_site':'E',
        'rand':'sjrand'
    }
    url = 'https://kyfw.12306.cn/passport/captcha/captcha-check'
    r = session.post(url, data=data)
    print(r.text)
    if '验证码校验失败' not in str(r.text):
        # 用户登录
        url = 'https://kyfw.12306.cn/passport/web/login'
        data = {
            'username': username,
            'password': password,
            'appid': 'otn'
        }
        r = session.post(url, data=data)
        print(r.text)
        if '密码输入错误' not in str(r.text):
            # 登录验证第一次请求
```

```
            url = 'https://kyfw.12306.cn/passport/web/auth/uamtk'
            data = {
                'appid': 'otn'
            }
            r = session.post(url, data=data)
            # 登录验证第二次请求
            newapptk = r.json()['newapptk']
            url = 'https://kyfw.12306.cn/otn/uamauthclient'
            data = {
                'tk': newapptk
            }
            r=session.post(url, data=data)
            print(r.text)
            return True
        else:
            return False
    return False
if __name__ =='__main__':
    # 创建持久化会话对象
    session = requests.session()
    username = 'xxxxxxx'
    password = 'xxxxxxx'
    login_info = login(username, password)
    print(session)
```

19.4 查 询 车 次

查询车次信息首先要输入出发地、目的地和出发日期，完成信息输入后，单击“查询”按钮，网站根据输入的信息返回相应的车次信息。

从一个正常的车次查询流程中发现，实际上网站与用户的交互是在用户单击“查询”按钮时发生的。在开发者工具捕捉到的请求如图 19-10 和图 19-11 所示。

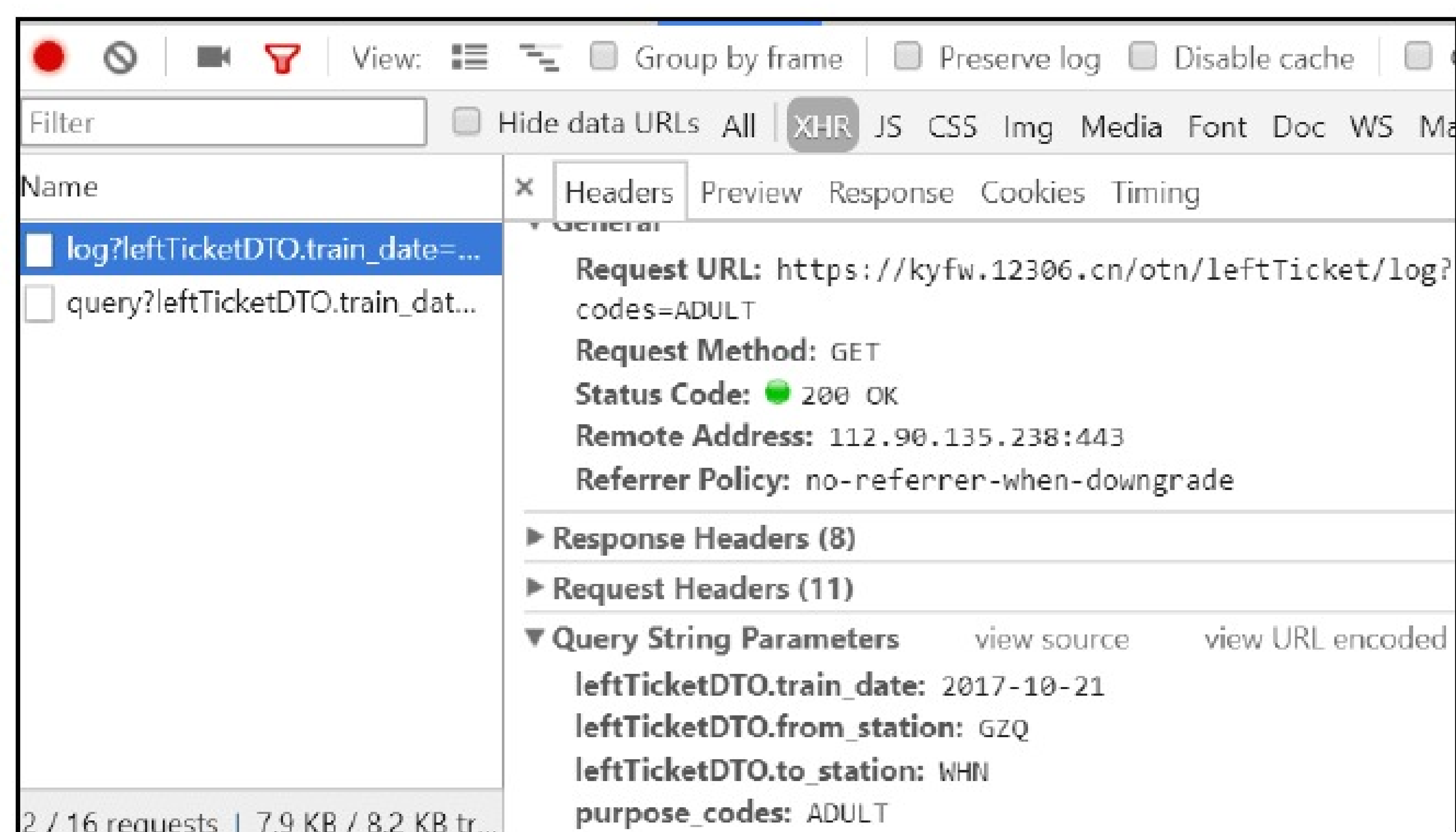


图 19-10　车票查询请求及响应内容一

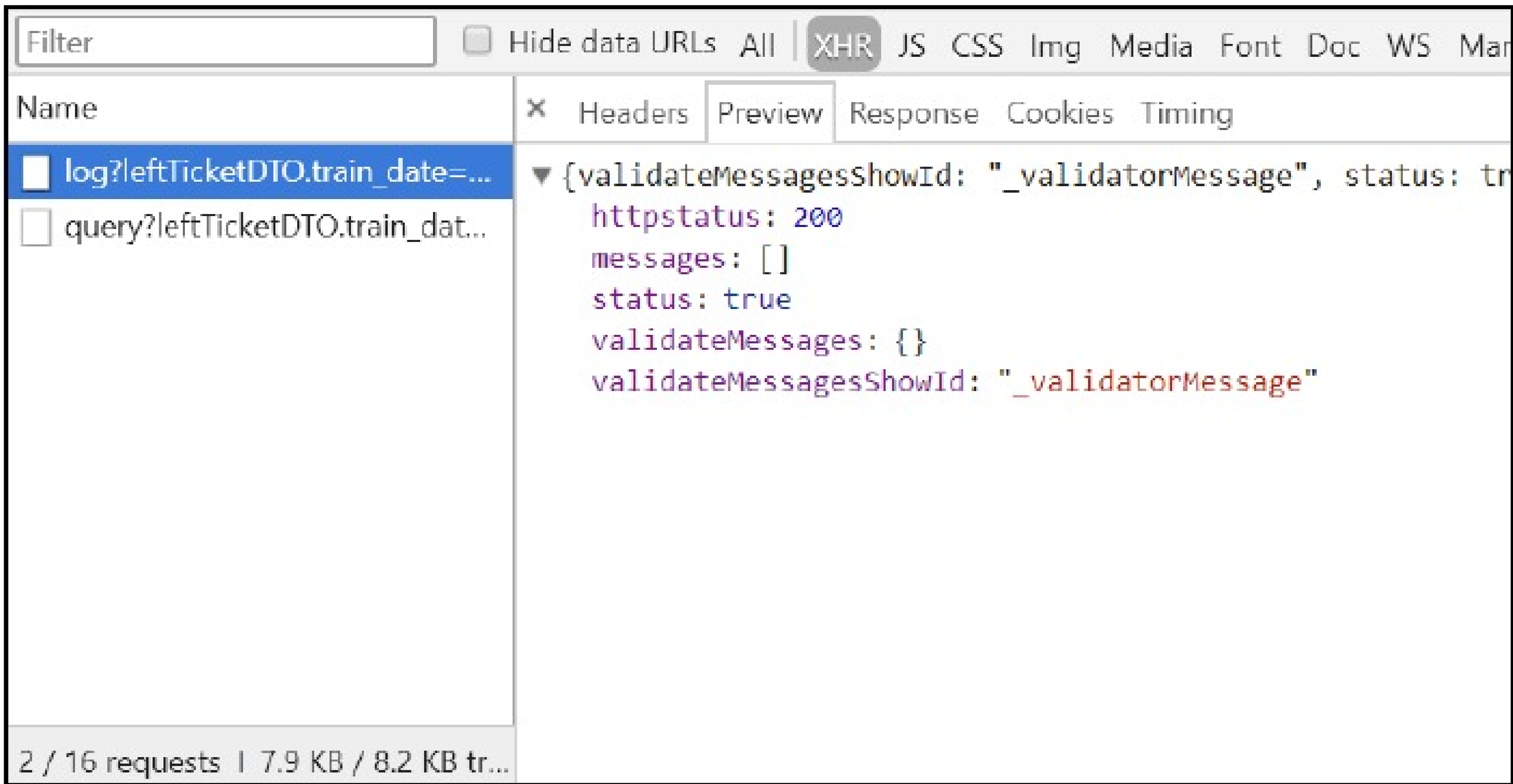


图 19-10（续）

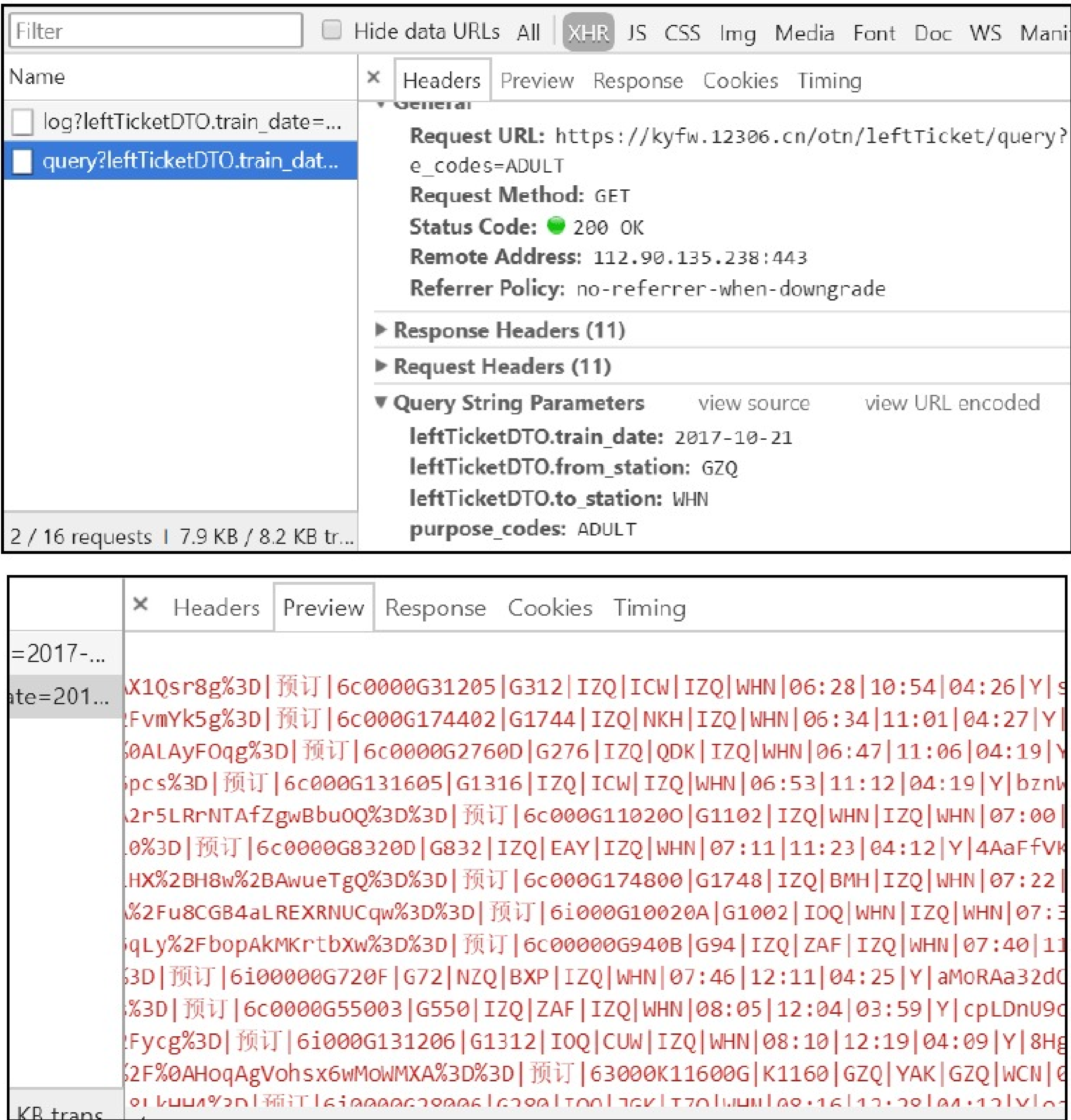


图 19-11　车票查询请求及响应内容二

对请求信息分析可知，用户在单击“查询”按钮后会发送两个 GET 请求，两个 GET 请求的参数是一致的。再查看两者的返回数据，图 19-10 的返回数据并没有太大用处，图 19-11 的返回数据内容较多，而且与网页显示的车次信息（见图 19-12）对比发现，两者的数据是可以相互匹配的。也就是说，网页上的车次信息由图 19-11 的请求信息生成并按照某种方式渲染到网页上。

车次	出发站/到达站	出发时间/到达时间	历时				
G312	始 广州南 过 武汉	06:28 10:54	04:26 当日到达	8	有	有	--
G1744	始 广州南 过 武汉	06:34 11:01	04:27 当日到达	5	1	有	--
G276	始 广州南 过 武汉	06:47 11:06	04:19 当日到达	20	有	有	--
G1316	始 广州南 过 武汉	06:53 11:12	04:19 当日到达	4	无	有	--
G1102	始 广州南 终 武汉	07:00 11:18	04:18 当日到达	5	14	有	--
G832	始 广州南 过 武汉	07:11 11:23	04:12 当日到达	18	有	有	--

图 19-12　查询车次信息

在代码中实现车次查询，首先要找到请求参数的数据来源。从图 19-10 和图 19-11 的参数可知，出发地和目的地都是英文字母，前两个字母是由城市名的拼音首字母组成的，最后一个字母无法确认。

注　意

图 19-11 的请求链接会不定时发生变更，有时候会将 URL 里面的“query”变为“queryA”，因此在编写代码的时候，需要设置两个不同的 URL 地址并分别对此发送 GET 请求，通过响应内容来确定正确的请求地址。

根据 19.3 节的请求参数查找方法，分别在 Doc、XHR、JS 标签查找和分析各个请求信息。我们在 JS 标签某个请求的响应内容中找到城市的字母编号，如图 19-13 所示。

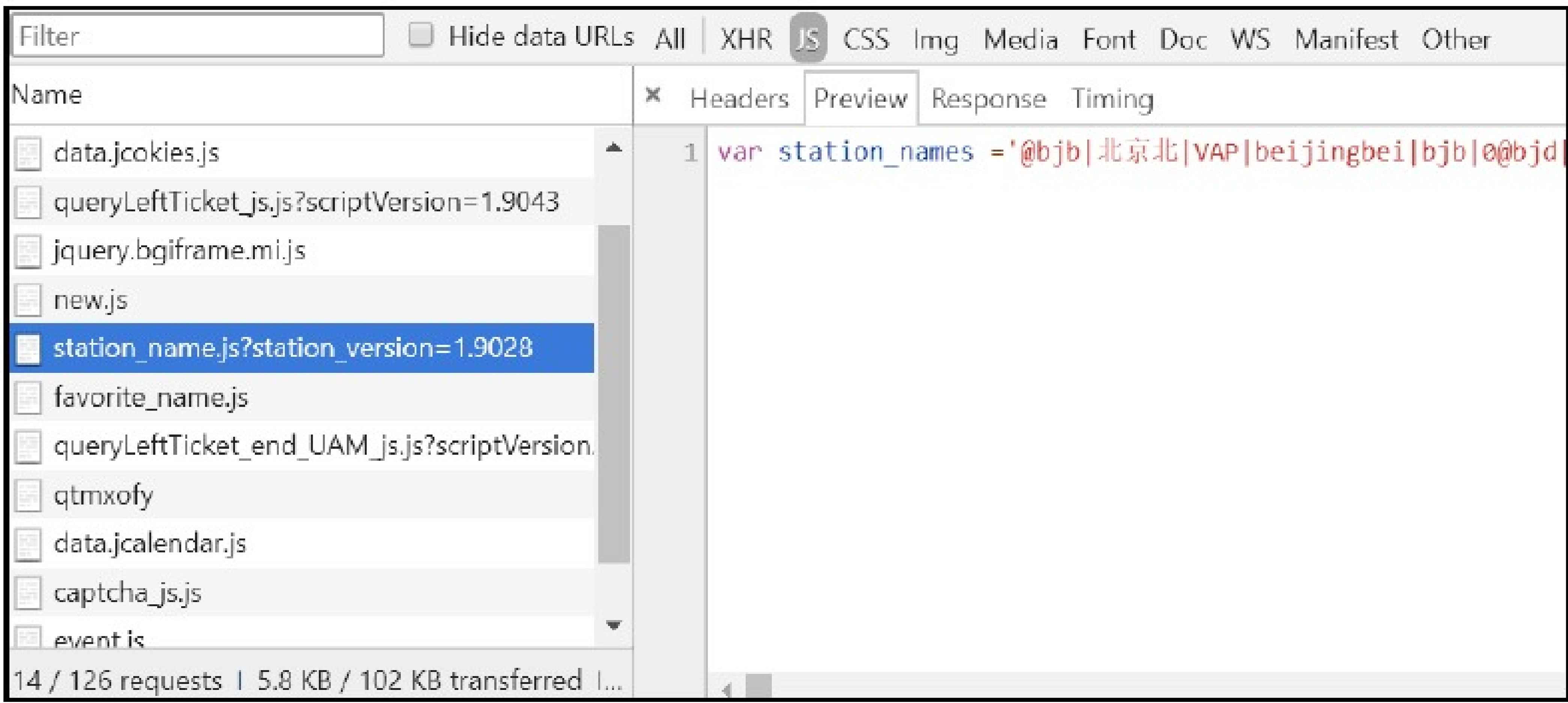


图 19-13　各个城市信息

观察其数据结构，发现每个城市之间以“@”为一个开始点，抽取部分内容进行分析：

```
@bjb|北京北|VAP|beijingbei|bjb|0@bjd|北京东|BOP|beijingdong|bjd|1@bji|北京
|BJP|beijing|bj|2@bjn|北京南|VNP|beijingnan|bjn|
```

在内容中可以找到城市名称和城市英文编号，分别是“北京北—VAP”“北京东—BOP”和“北京—BJP”。每个数据之间以“|”隔开，而我们所需的数据分别是含有“@”的数据后的第一位和第二位。根据这个规律，实现代码如下：

```
import requests
```

```
def city name():
    url = 'https://kyfw.12306.cn/otn/resources/js/framework/
            station name.js?station version=1.9031'
    city_code = session.get(url)
    city code list = city code.text.split("|")
    city dict = {}
    for k, i in enumerate(city_code_list):
        if '@' in i:
            # 城市名作为字典的键，城市编号作为字典的值
            city dict[city code list[k + 1]] = city code list[k + 2]
    return city_dict
```

现在得到图 19-11 的请求参数，接下来对图 19-11 返回的车次信息进行清洗，获取我们所需的数据。在实际中，每班车次存在两种情况，分别是有余票和无票。对这两种情况的数据进行分析和对比：

```
"%2BJ1GyMIoe1W3AmIOyCC0ho%2FthJG%2F4PfGMtxEKS2Byrl9JuOTLHBVrBJfd40RUtRiP%2
BbVdnpdTiob%0AjYQA1VkkqS16P5EsPeuK%2Fldya6KLswYyo%2BBwsLXQkr4i2D2tDAndyjyh
OOhMZEKn%2BNFAZaiBMRQi%0ATRBqKt8pYlNmBeu9lgEQdsdvMJ23SGTzzptyCwYtmEut4Ffog
6LPkywCZT1EfeFOO4Hp%2BU%2BF2NIk%0AeeFN8fY%3D|预订|6c0000G31205|G312|IZQ|ICW
|IZQ|WHN|06:28|10:54|04:26|Y|%2FCaZQBZdKPD7OTjFodC1%2F7y2N8jqdmZRAJH0rECZr
eKGar1Z|20171023|3|QZ|01|09|null|0|||||||||||有|有|8||O0M090|OM9"
```

上述数据是图 19-12 G312 车次的列车信息，每个数据由“|”连接组成一条完整的车次信息。有一部分数据与图 19-12 对得上，其余的数据暂时无法确定，可能会在后续的流程中起到重要作用。我们再抽取无票的车次信息，如图 19-14 所示。

G542	过 广州南 过 武汉	09:44 14:01	04:17 当日到达	无	2	无	--	--	--	--	--	--	--	--	预订
K770	始 广州 过 武昌	10:00 23:41	13:41 当日到达	--	--	--	--	4	--	有	--	有	有	--	预订
G66	始 广州南 过 武汉	10:00 13:38	03:38 当日到达	无	无	无	--	--	--	--	--	--	--	--	预订
K1348	过 广州 过 武昌	10:06 23:23	13:17 当日到达	--	--	--	--	10	--	有	--	有	有	--	预订
G1006	过 广州南 终 武汉	10:11 14:17	04:06 当日到达	3	2	5	--	--	--	--	--	--	无	--	预订

图 19-14　无票的车次信息

```
"|预订|6c00000G6605|G66|IZQ|BXP|IZQ|WHN|10:00|13:38|03:38|N|8zmC3adDZKJi
1D3WPp2sMDr83uz91FxKN%2FYEoN1IG%2FD7AaME|20171023|3|Q9|01|03|0|0||||||
||||无|无|无||O0M090|OM9"
```

可以看到，G66 车次已经满座无票，分析其数据内容发现“预订”前面的数据为空，其他信息和有余票的车次信息大致相同。说明“预订”前面的数据可以区分车次是否还有余票。

无论是无票还是有余票，都是以“|”将各个信息连接起来组成一条车次信息，按照这个规律，车次信息清洗代码如下：

```
train info info = r.json()
train_info_dict = {}
for i in train info info['data']['result']:
train info status = i.split('|')
if train info status[0] != '':
        train info dict['secretStr'] = train info status[0]
        train_info_dict['train_no'] = train_info_status[2]
```

```
        train_info_dict['stationTrainCode'] = train_info_status[3]
        train_info_dict['fromStationTelecode'] = train_info_status[4]
        train_info_dict['toStationTelecode'] = train_info_status[7]
        train_info_dict['leftTicket'] = train_info_status[12]
        train_info_dict['train_location'] = train_info_status[15]
```

train_info_info 是图 19-11 返回的响应内容，train_info_info['data']['result']是直接定位车次信息，然后对车次信息以“|”分割，得到新的列表，通过判断“预订”前面的数据是否为空，排除无票的车次信息。

综合上述分析，将获取城市编号和获取车次信息的代码整合优化，得到如下代码：

```
import requests
# 获取城市编号
def city_name():
    url = 'https://kyfw.12306.cn/otn/resources/js/
            framework/station_name.js?station_version=1.9031'
    city_code = session.get(url)
    city_code_list = city_code.text.split("|")
    city_dict = {}
    for k, i in enumerate(city_code_list):
        if '@' in i:
            # 城市名作为字典的键，城市英文编号作为字典的值
            city_dict[city_code_list[k + 1]] = city_code_list[k + 2]
    return (city_dict)
# 获取车次信息
def train_info(train_date, query_from_station_name, query_to_station_name):
    # 调用函数 city_name 获取城市编号
    city_dict = city_name()
    from_station = city_dict[query_from_station_name]
    to_station = city_dict[query_to_station_name]
    # 获取车次信息
    while 1:
        # 第一次请求
        url = 'https://kyfw.12306.cn/otn/leftTicket/log?
                leftTicketDTO.train_date=%s&leftTicketDTO.from_station=
                %s&leftTicketDTO.to_station=%s&purpose_codes=ADULT'
                % (train_date, from_station, to_station)
        r = session.get(url)
        # 第二次请求
        # 请求地址的 query 可能变为 queryA，可通过 try……except 控制
        try:
            url = 'https://kyfw.12306.cn/otn/leftTicket/query?
                    leftTicketDTO.train_date=%s&leftTicketDTO.from_station=
                    %s&leftTicketDTO.to_station=%s&purpose_codes=ADULT'
                    % (train_date, from_station, to_station)
            r = session.get(url)
            test = r.json()['data']['result']
        except:
            url = 'https://kyfw.12306.cn/otn/leftTicket/queryA?
                    leftTicketDTO.train_date=%s&leftTicketDTO.from_station=
                    %s&leftTicketDTO.to_station=%s&purpose_codes=ADULT'
                    % (train_date, from_station, to_station)
            r = session.get(url)
        time.sleep(2)
```

```
        if '非法请求' not in str(r.text) and '"result":[]' not in str(r.text):
            train_info_info = r.json()
            train_info_dict = {}
            for i in train_info_info['data']['result']:
                train_info_status = i.split('|')
                if train_info_status[0] != '':
                    train_info_dict['secretStr'] = train_info_status[0]
                    train_info_dict['train_no'] = train_info_status[2]
                    train_info_dict['stationTrainCode'] = train_info_status[3]
                    train_info_dict['fromStationTelecode'] =
                                                    train_info_status[
4]
                    train_info_dict['toStationTelecode'] = train_info_status[7]
                    train_info_dict['leftTicket'] = train_info_status[12]
                    train_info_dict['train_location'] = train_info_status[15]
                    return train_info_dict
if __name__=='__main__':
    session = requests.session()
    username = '13435423143'
    password = 'XXXXXXXX'
    login_info = login(username,password)
    # 判断是否登录成功
    if login_info:
        train_date = 'YYYY-MM-DD'
        query_from_station_name = '广州'
        query_to_station_name = '武汉'
        train_info_dict = train_info(train_date,query_from
                        station_name,query_to_station_name)
```

函数 train_info()定义了三个参数：

（1）train_date 为出发时间，日期格式为 YYYY-MM-dd。
（2）query_from_station_name 为出发地，以城市中文名作为函数参数，如“广州”。
（3）query_to_station_name 为目的地，以城市中文名作为函数参数，如“武汉”。

函数整体的开发逻辑大致如下：

（1）调用函数 city_name()获取城市编号，将出发地和目的地的中文转换成英文编号。
（2）使用 while 循环获取车次信息，目的是保证车次信息获取成功，因为在浏览器上每次查询车票不一定会返回车次信息，循环的作用相当于用户不停地单击“查询”按钮。每次循环设置延时 2 秒，这个等待时间是为了防止程序访问太过频繁而被网站认为是机器人。
（3）判断第二次请求所返回的响应内容是不是车次信息，若是，则获取第一条有余票的车次信息并返回，反之一直循环发送请求，直到获取为止。

运行上述代码，结果如图 19-15 所示。

```
请输入验证码：3
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certific
  InsecureRequestWarning)
{"result_message":"验证码校验成功","result_code":"4"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certific
  InsecureRequestWarning)
{"result_message":"登录成功","result_code":0,"uamtk":"K3DHvIKb1TwmXOn291Mj1u1MQI8dZrrhonvwsw52kPI921110"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certific
  InsecureRequestWarning)
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certific
  InsecureRequestWarning)
{"apptk":"OOhKe__-rT1OOYcuL9mOOt4Vzy8oRAmjij2Uz5wLyOUjn1110","result_code":0,"result_message":"验证通过","username":"黄永祥"}
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certific
  InsecureRequestWarning)
F:\Python\lib\site-packages\urllib3\connectionpool.py:852: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certific
  InsecureRequestWarning)
{'stationTrainCode': 'Z122', 'leftTicket': 'vLPdZKMYTFOZhyDbhCg6ZQaECB1tOU2pxNoTsf6vUoAgud%2FvHQF2CCFHw0g%3D', 'train_no': '630000Z12208',
```

图 19-15　车票查询结果

19.5　预 订 车 票

完成车次查询，接下来实现车票预订功能。由于在 19.4 节中，函数 train_info()只返回第一条有余票的车次信息，以图 19-12 为例，如果车次 G312 有余票，就直接返回该车次信息，如果满座无票，就取下一班车次信息再判断是否有余票，直到取得有余票的车次为止。

我们对图 19-12 的 G312 车次进行车票预订，单击“预订”按钮进行分析，发现单击按钮会触发一个 302 跳转，在跳转前，截取到两个 POST 请求，如图 19-16 和图 19-17 所示。

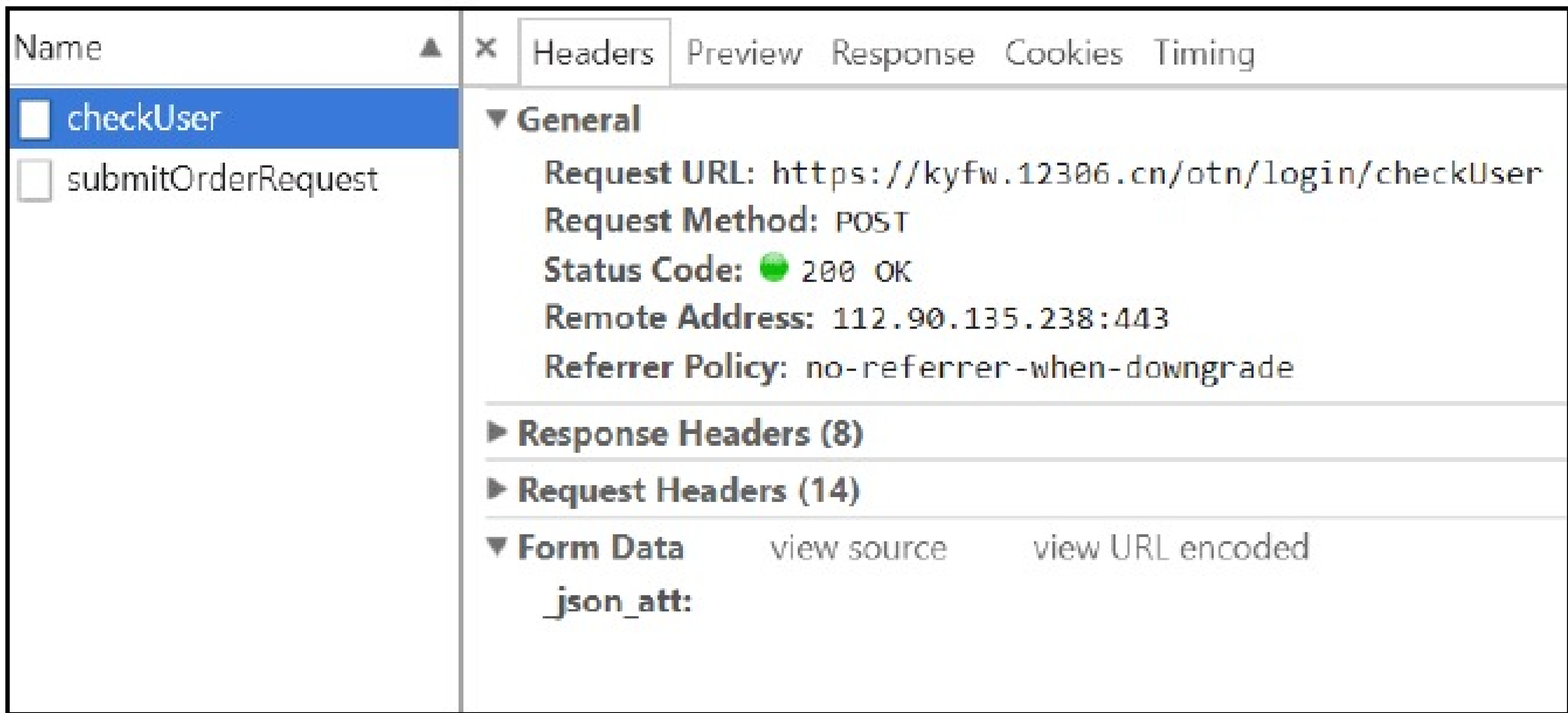


图 19-16　车票预订请求一

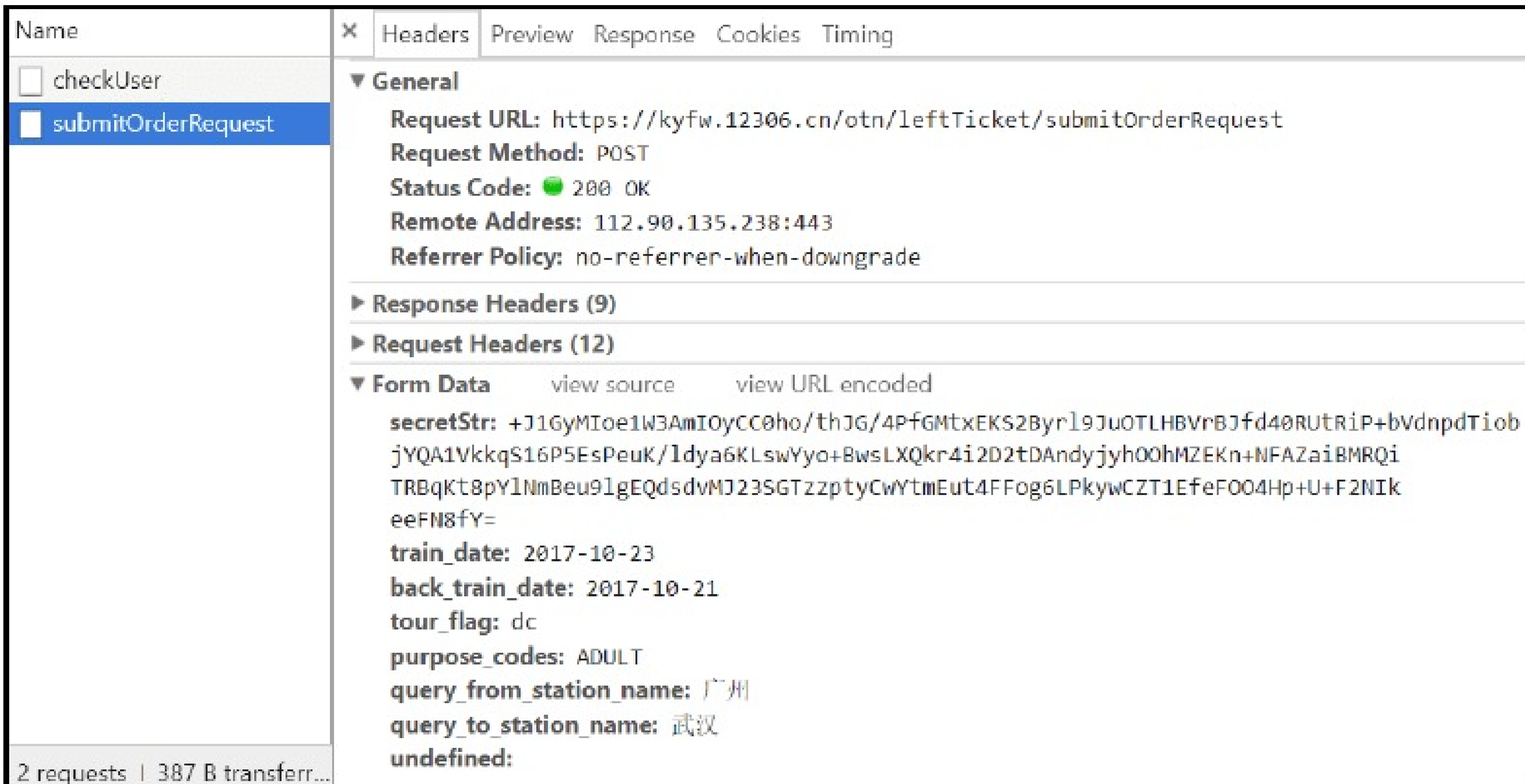


图 19-17　车票预订请求二

从图 19-16 的请求链接分析可知，这是用于检查用户登录状态，请求参数的数据为空。从图 19-17 的请求链接、请求参数分析得知，这是一个提交订单的请求，请求参数分析如下：

（1）train_date、query_from_station_name 和 query_to_station_name 在 19.4 节已明确知道。

（2）back_train_date 是回程的日期。如果单程订票，该参数直接取当天日期即可。

（3）tour_flag 从参数值（dc）判断，这是由“单程”拼音的首字母组成的。

（4）purpose_codes 的参数值固定不变；undefined 的参数值为空。

（5）SecretStr 是一串不规则的数据，回顾 19.4 节，在车次信息里面也含有不规则数据，猜想这两者是否一样，我们抽取 G312 车次的信息，如下所示：

```
"%2BJ1GyMIoe1W3AmIOyCC0ho%2FthJG%2F4PfGMtxEKS2Byrl9JuOTLHBVrBJfd40RUtRiP%2
BbVdnpdTiob%0AjYQA1VkkqS16P5EsPeuK%2Fldya6KLswYyo%2BBwsLXQkr4i2D2tDAndyjyh
OOhMZEKn%2BNFAZaiBMRQi%0ATRBqKt8pYlNmBeu9lgEQdsdvMJ23SGTzzptyCwYtmEut4Ffog
6LPkywCZT1EfeFOO4Hp%2BU%2BF2NIk%0AeeFN8fY%3D|预订|6c0000G31205|G312|IZQ|ICW
|IZQ|WHN|06:28|10:54|04:26|Y|%2FCaZQBZdKPD7OTjFodC1%2F7y2N8jqdmZRAJH0rECZr
eKGar1Z|20171023|3|QZ|01|09|null|0|||||||||||有|有|8||O0M090|OM9"
```

通过对比发现，两者的数据是一样的，只不过某些特殊符号经过了编码处理，如“+”变成“%2B”、“/”变成“%2F”。在第 5.7 节中，我们已介绍了 urllib.parse 提供了函数对数据的编码和解码处理。

综合上述分析，车票预订功能代码如下：

```
    def train order(secretStr, train date, query from station name,
query to station name):
        # 获取当前日期
        back train date = datetime.datetime.now().strftime('%Y-%m-%d')
        # 用户登录检查
        url = 'https://kyfw.12306.cn/otn/login/checkUser'
        data = {
            ' json att': ''
        }
        r = session.post(url, data=data)
        # 提交车票预订请求
        url = 'https://kyfw.12306.cn/otn/leftTicket/submitOrderRequest'
        data = {
            'secretStr': secretStr,
            'train date': train date,
            'back train date': back train date,
            'tour flag': 'dc',
            'purpose codes': 'ADULT',
            'query from station name': query from station name,
            'query to station name': query to station name,
            'undefined': ''
        }
        r = session.post(url, data=data)

    if  name  ==' main  ':
        session = requests.session()
        username = '13435423143'
        password = 'xxxxxxxxxxxx'
        login_info = login(username, password)
```

```
    if login_info:
        train_date = '2017-10-23'
        query_from_station_name = '广州'
        query_to_station_name = '武汉'
        train_info_dict = train_info(train_date,
                      query_from_station_name,query_to_station_name)
        # 数据格式化处理
        secretStr=parse.unquote(train_info_dict['secretStr'])
        train_order(secretStr, train_date,
                    query_from_station_name, query_to_station_name)
```

19.6　提 交 订 单

车票预订提交之后，从浏览器上可以看到页面发生了跳转，在新的页面上需要填写乘客信息，最后提交订单，如图 19-18 所示。

图 19-18　填写乘客信息

填写好乘客信息之后，单击“提交订单”按钮，通过开发者工具捕捉请求信息，如图 19-19 和图 19-20 所示。

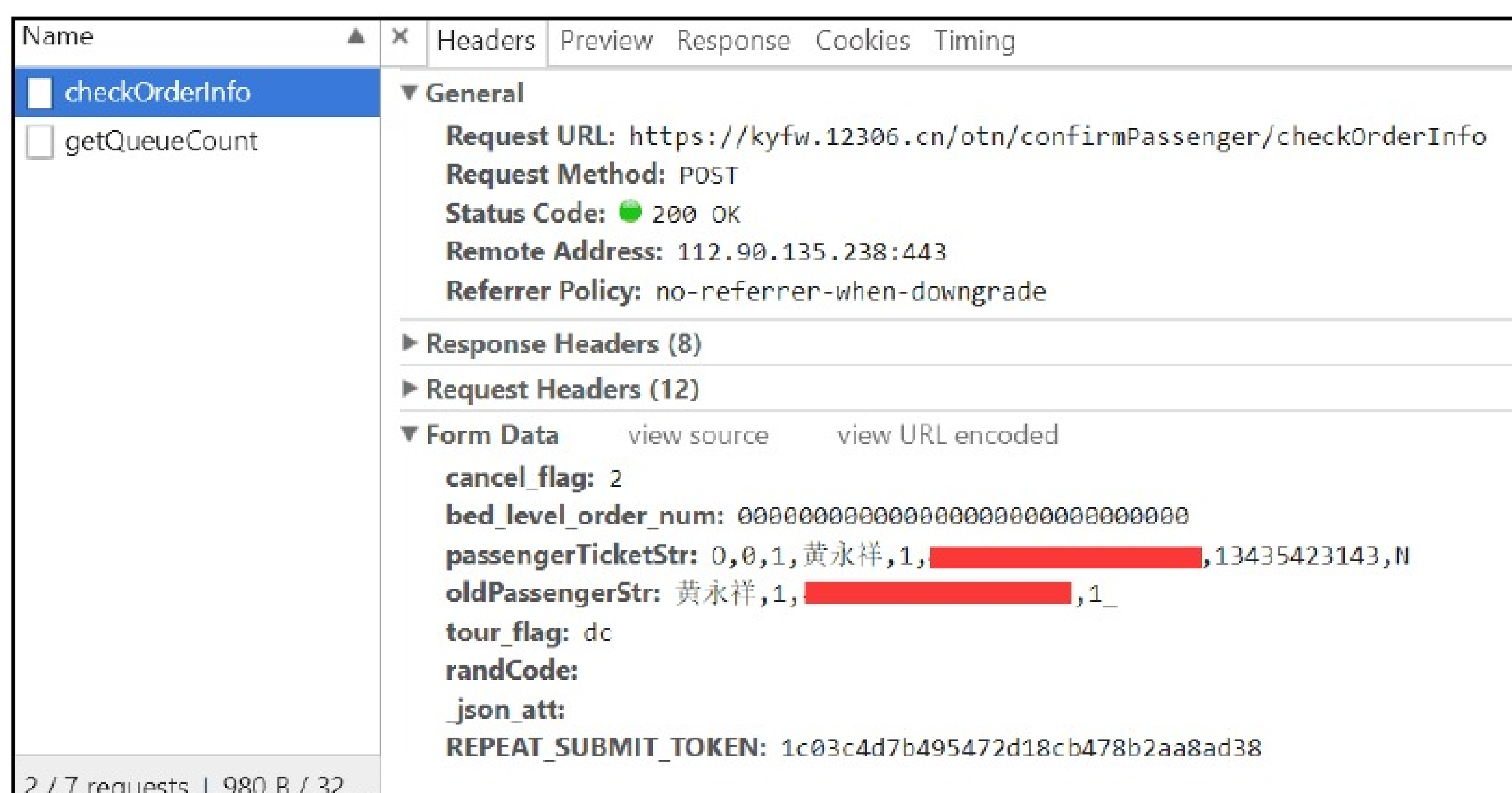


图 19-19　提交订单请求一

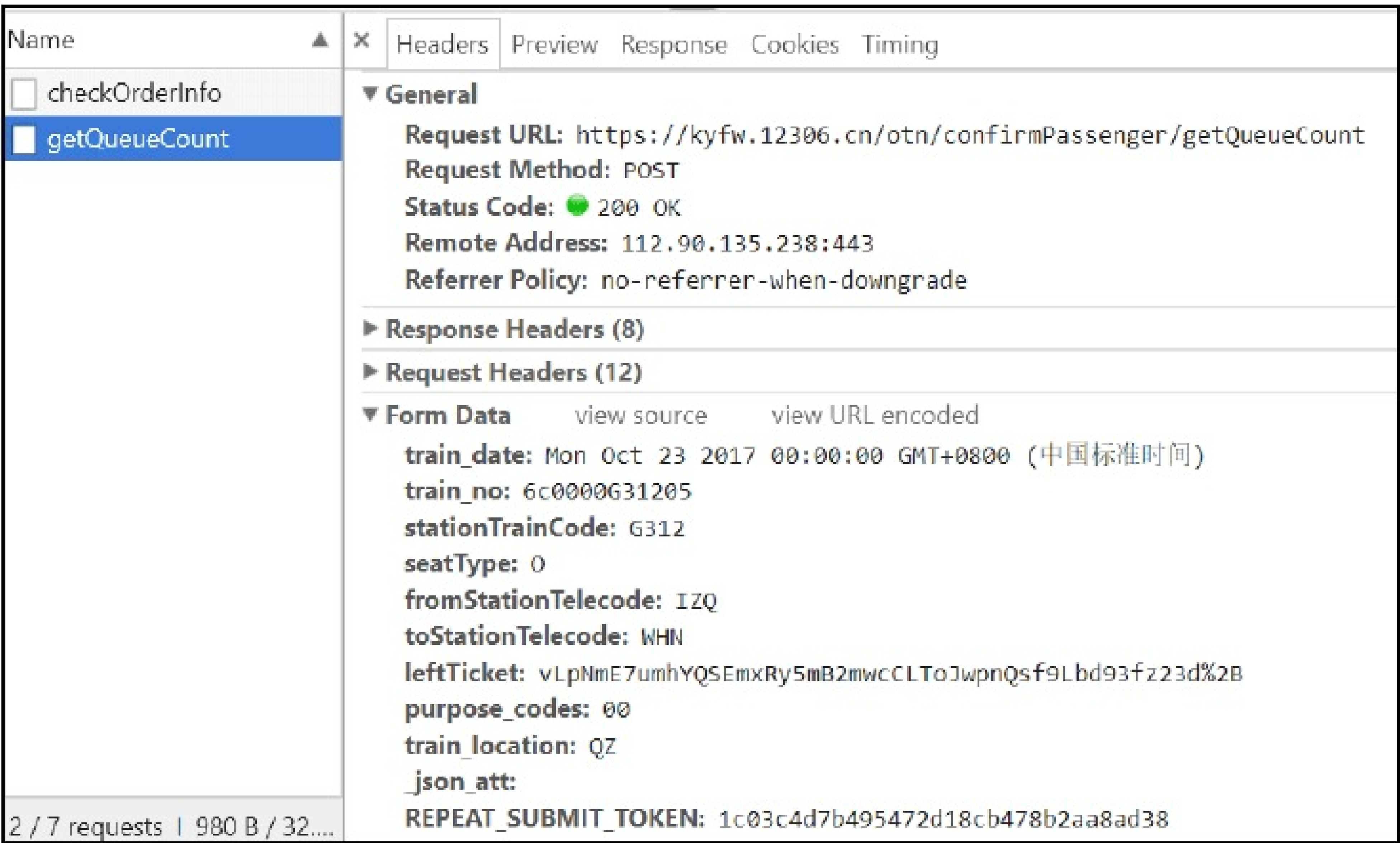


图 19-20　提交订单请求二

单击“提交订单”按钮后，Chrome 捕捉到两个 POST 请求。从图 19-19 的请求参数来看：

（1）参数 cancel_flag、bed_level_order_num、tour_flag、randCode 和_json_att 是固定不变的。

（2）参数 REPEAT_SUBMIT_TOKEN 无法得知，可能由其他请求生成。

（3）参数 passengerTicketStr 和 oldPassengerStr 代表个人乘车信息。观察参数组成，发现每个数据之间用逗号分隔，每个数据有可能代表某个意思，为了进一步验证数据的意义，我们修改乘客信息，如图 19-21 所示。

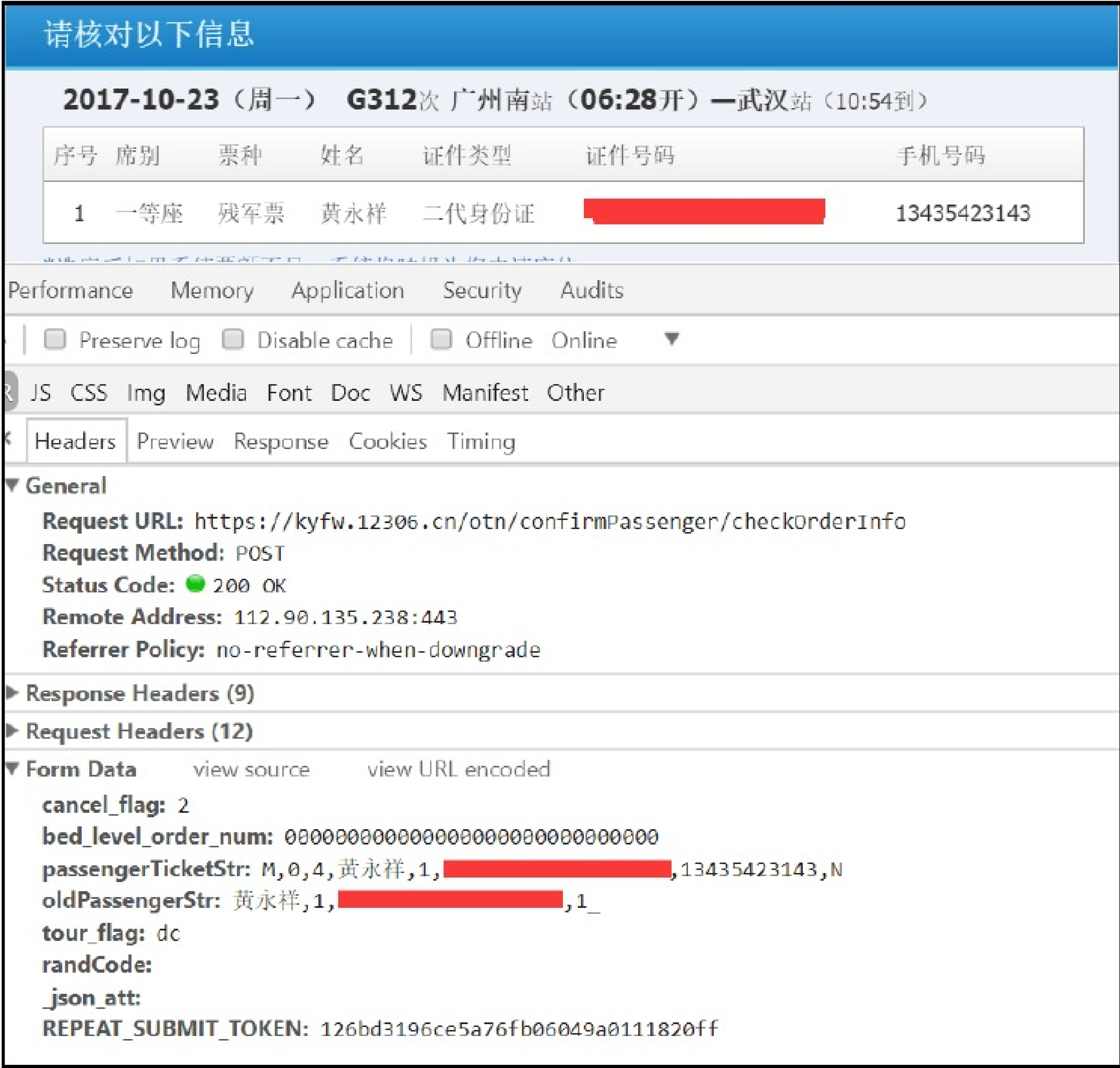


图 19-21　验证请求参数

对比图 19-19 和图 19-21 发现，oldPassengerStr 的数据不会随着席别和票种的变化而变化，而

passengerTicketStr 会随之变化。

为了寻找 passengerTicketStr 的变化规律，我们多次修改乘客信息，发现变化规律如下：

（1）passengerTicketStr 前三个数字 M、0、2 的 M 和 2 分别代表一等座和儿童票，0 是固定不变的。

（2）席别编号：软卧=4、硬座=1、硬卧=3、二等座= O（字母 O）、一等座=M、商务座=9。

（3）票种编号：成人票=1、儿童票=2、学生票=3、残军票=4。

确定了参数 passengerTicketStr 的数据含义，同时发现一个问题，每班车次的席别信息是动态变化的，但无法确定当前车次还剩下哪些席别可供我们选择。从图 19-18 看到，席别信息是一个下拉框控件，里面含有剩余的席别信息，因此需要找出当前车次剩余的席别信息，用于构建参数 passengerTicketStr。分别从 XHR、JS 和 Doc 标签查找席别信息，最终在 Doc 标签找到，如图 19-22 和图 19-23 所示。

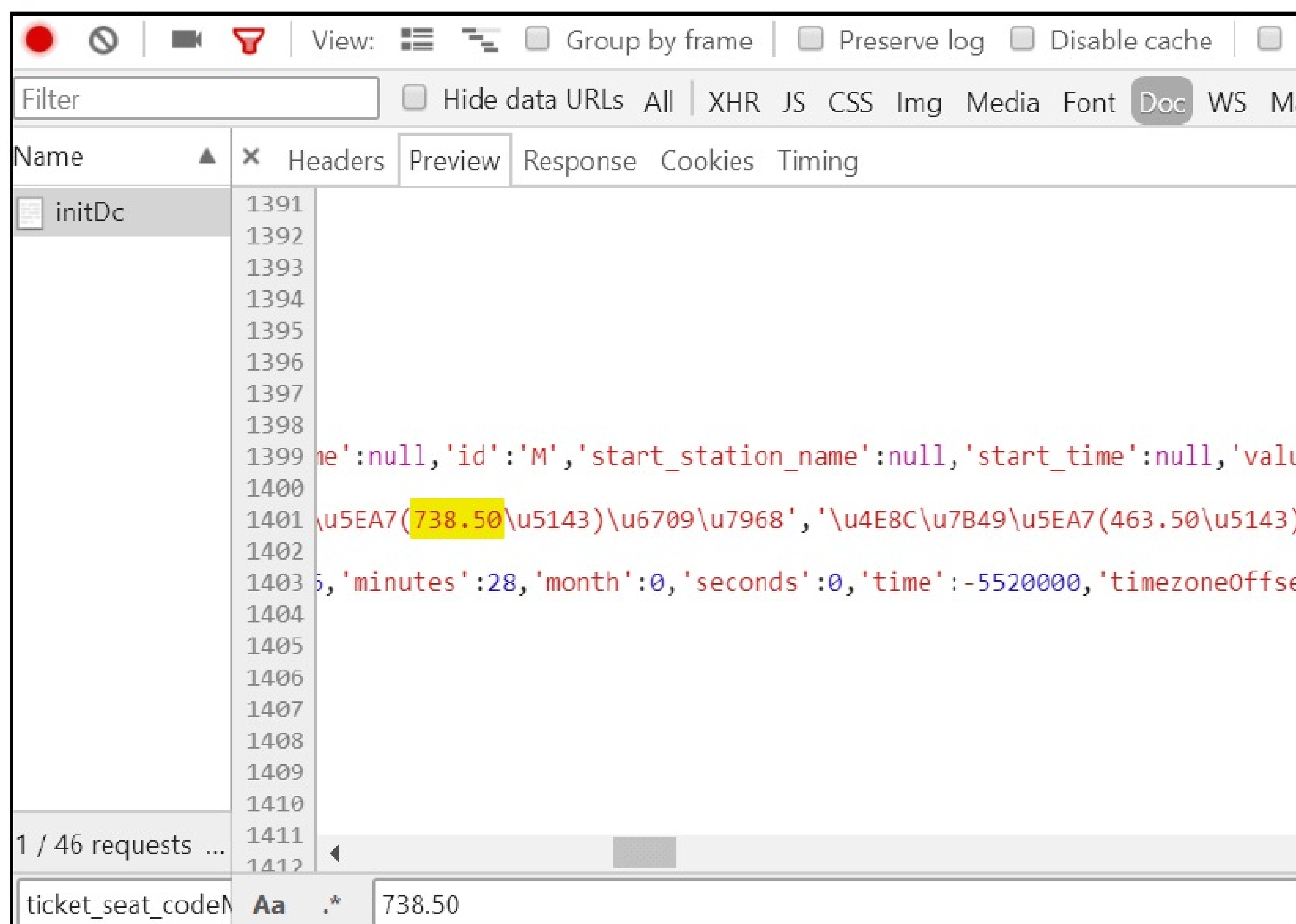


图 19-22　查找席别信息一



图 19-23　查找席别信息二

从图 19-22 和图 19-23 看到，由于席别的价钱具有特殊唯一性，因此可利用其特殊性来实现快速查找，发现在 Doc 的请求信息中找到剩余的席别信息。席别信息写在变量 ticket_seat_codeMap 中，以“id：X”格式存放。

再回到图 19-19 的 REPEAT_SUBMIT_TOKEN 参数，从内容中无法得知数据含义，而且数据是动态变化的，为了确认数据来源，分别从 XHR、JS 和 Doc 标签进行排查，在 Doc 标签找到数据来源，如图 19-24 所示。

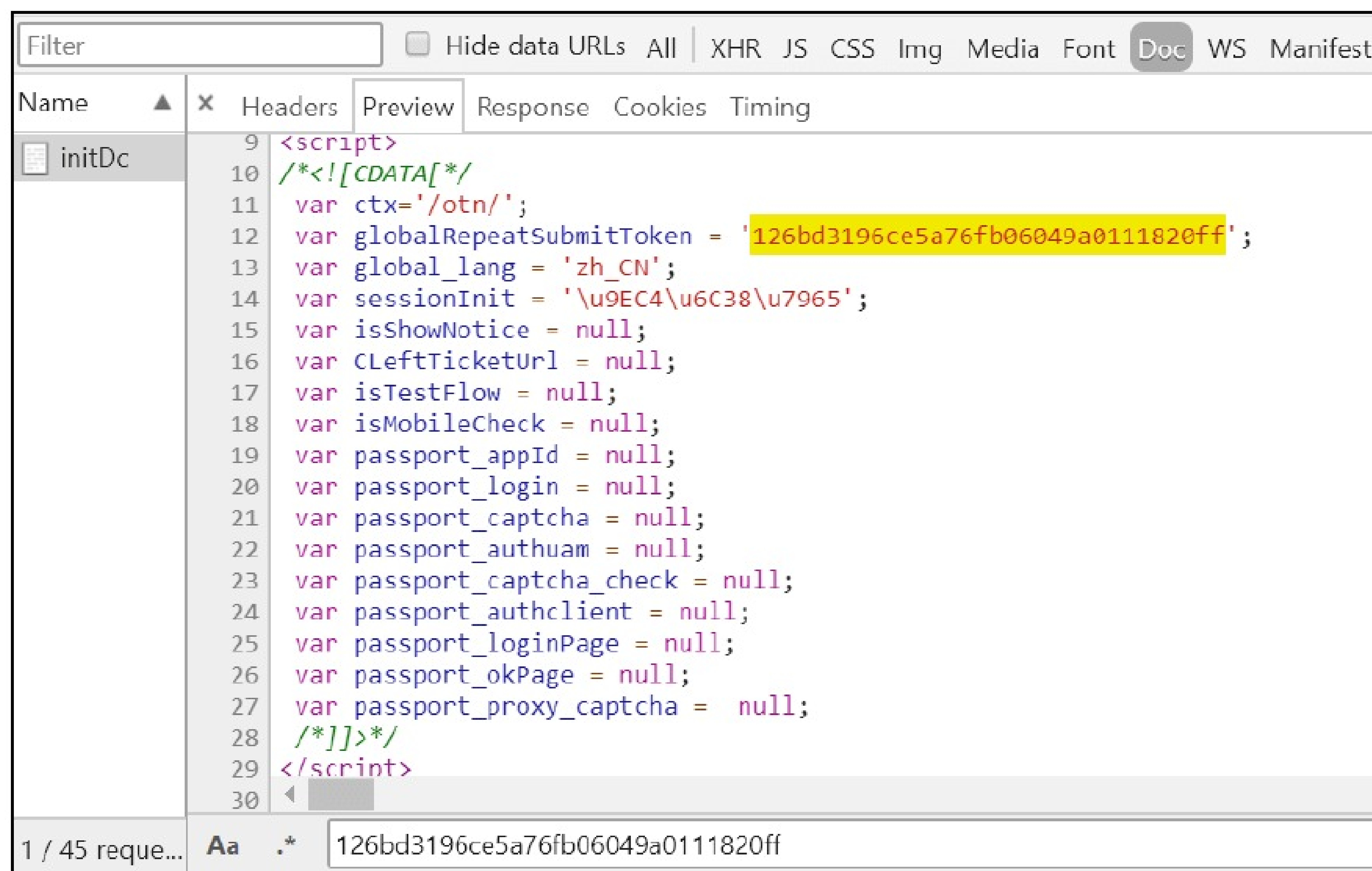


图 19-24　查找请求参数

可以发现，参数 REPEAT_SUBMIT_TOKEN 是 Doc 标签的 JavaScript 变量。综合上述分析，图 19-19 的实现代码如下：

```
# 获取 Doc 标签的数据
url = 'https://kyfw.12306.cn/otn/confirmPassenger/initDc'
data = {
' json att': '',
}
r = session.post(url, data=data)
# 获取参数
get token = r.text.split('globalRepeatSubmitToken')[1].split(';')[0].
replace('=','').replace("'",'').strip()
seat_code_str = r.text.split('ticket_seat_codeMap=')[1].
                split(';')[0].strip()
# 找出座位编号并去重
temp list = re.findall(r"'id':'(.+?)',",seat code str)
temp list = list(set(temp list))
# 获取第一个席别编号
seatType = temp list[0]
# 检查订单信息
# 构建请求参数，name-乘客姓名，identity_card-身份证号
# phone number-电话号码，票种为成人票
oldPassengerStr = name + ',1,' + identity_card + ',1_'
passengerTicketStr = seatType + ',0,1,' + name + ',1,' + identity_card + ','
```

```
+ phone number + ',N'
    url = 'https://kyfw.12306.cn/otn/confirmPassenger/checkOrderInfo'
    data = {
    'cancel_flag': '2',
    'bed level order num': '000000000000000000000000000000',
    'passengerTicketStr': passengerTicketStr,
    'oldPassengerStr': oldPassengerStr,
    'tour flag': 'dc',
    'randCode': '',
    ' json att': '',
    'REPEAT SUBMIT TOKEN': get token
    }
    r = session.post(url,data=data)
```

上述代码只实现了图 19-19 的功能，要完成订单提交，还要实现图 19-20 的请求，图 19-20 请求的参数如下：

（1）train_date、train_no、stationTrainCode、fromStationTelecode、leftTicket 和 train_location 可在 15.4 节的车次信息中获取。

（2）REPEAT_SUBMIT_TOKEN 和 seatType 可在图 19-19 实现的代码中获取。

（3）purpose_codes 和_json_att 是固定不变的。

结合图 19-19 和图 19-20 的分析，提交订单的功能代码如下：

```
    def creat order(name, identity card, phone number, train date,
train info dict):
        # 获取 Doc 标签的数据
        url = 'https://kyfw.12306.cn/otn/confirmPassenger/initDc'
        data = {
            ' json att': ''
        }
        r = session.post(url, data=data)
        # 获取参数
        key check isChange = r.text.split('key check isChange')[1].
                        split(',')[0].replace(':', '').replace("'", '').strip()
        get token = r.text.split('globalRepeatSubmitToken')[1].
                        split(';')[0].replace('=', '').replace("'", '').strip()
        seat code str = r.text.split('ticket seat codeMap=')[1].
                        split(';')[0].strip()
        # 找出席别编号并去重
        temp list = re.findall(r"'id':'(.+?)',", seat code str)
        temp list = list(set(temp list))
        seatType = temp list[1]

        # 检查订单信息
        # 构建请求参数，name-乘客姓名，identity_card-身份证号
        # phone number-电话号码，票种为成人票
        oldPassengerStr = name + ',1,' + identity card + ',1 '
        passengerTicketStr = seatType+',0,1,'+name+',1,'+identity card+
                            ',' + phone_number + ',N'
        url = 'https://kyfw.12306.cn/otn/confirmPassenger/checkOrderInfo'
        data = {
            'cancel_flag': '2',
```

```
            'bed level order num': '000000000000000000000000000000',
            'passengerTicketStr': passengerTicketStr,
            'oldPassengerStr': oldPassengerStr,
            'tour_flag': 'dc',
            'randCode': '',
            ' json att': '',
            'REPEAT_SUBMIT_TOKEN': get_token
        }
        r = session.post(url, data=data)
        # 提交订单信息
        # train date，train no,stationTrainCode
        # fromStationTelecode, toStationTelecode,
        # leftTicket,train_location 来自车次信息
        # seatType 和 REPEAT_SUBMIT_TOKEN 来自 Doc 标签的数据
        # purpose_codes 和 _json_att 固定不变
        while 1:
            url = 'https://kyfw.12306.cn/otn/confirmPassenger/getQueueCount'
            #日期格式化处理
            check ticket date = train date + ' 00:00:00'
            timeArray = time.strptime(check ticket date, "%Y-%m-%d %H:%M:%S")
            date = time.strftime("%a %b %d %Y", timeArray)
            data = {
                'train_date': date + ' GMT+0800 (中国标准时间)',
                'train no': train info dict['train no'],
                'stationTrainCode': train info dict['stationTrainCode'],
                'seatType': seatType,
                'fromStationTelecode': train_info_dict['fromStationTelecode'],
                'toStationTelecode': train info dict['toStationTelecode'],
                #leftTicket 进行数据格式化处理
                'leftTicket': parse.unquote(train info dict['leftTicket']),
                'purpose codes': '00',
                'train location': train info dict['train location'],
                ' json att': '',
                'REPEAT SUBMIT TOKEN': get token
            }
            r = session.post(url, data=data)
            print(r.text)
            # 判断请求是否成功
            if '系统繁忙，请稍后重试' not in str(r.text):
                break
    if   name   ==' main  ':
        session = requests.session()
        username = '13435423143'
        password = 'xxxxx'
        login info = login(username,password)
        if login info:
            train date = '2017-10-23'
            query from station name = '广州'
            query_to_station_name = '武汉'
            train info dict = train info(train date,
                  query from station name,query to station name)
            secretStr = parse.unquote(train info dict['secretStr'])
            train order(secretStr, train date, query from station name,
                          query_to_station_name)
```

```
        name = '黄永祥'
        identity card = 'xxxxxxxxxxxxx'
        phone number = '13435423143'
        creat_order(name, identity_card, phone_number, train_date,
                    train_info_dict)
```

19.7　生 成 订 单

用户提交订单之后，下一步是确认订单，在网页上单击“提交订单”按钮，会弹出信息核对窗口，主要供用户确认乘车信息和乘客的个人信息，如图 19-25 所示。

图 19-25　信息核对

当用户信息核对无误后，单击“确认”按钮，订单就会自动生成，同时在开发者工具捕捉到两个请求信息，如图 19-26 所示。

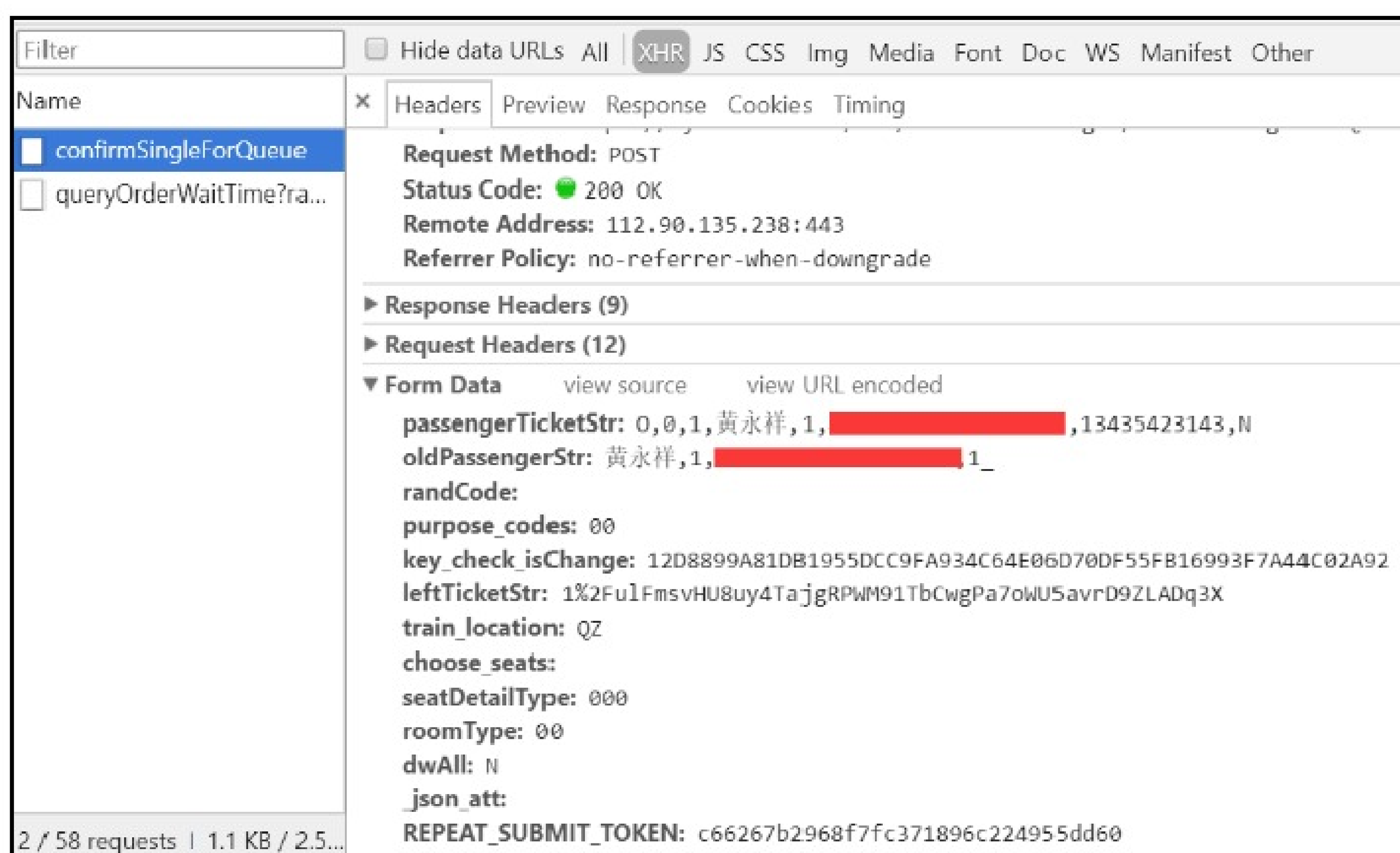


图 19-26　确认订单

从图 19-26 的请求参数分析，参数分为三种类型：

（1）已明确的参数，在前面的内容中已分析说明，如 passengerTicketStr、oldPassengerStr、leftTicket、train_location 和 REPEAT_SUBMIT_TOKEN。

（2）参数值是固定不变的，如 randCode、purpose_codes、seatDetailType、roomType、dwAll 和_json_att。

（3）参数无法明确，如 choose_seats 和 key_check_isChange。

从参数 choose_seats 的命名分析，其代表选座信息。在图 19-25 中，用户确认订单之前，还可以选择座位位置，座位以 A～F 命名，如果参数值为空，就默认为网站自动选座；如果参数值为 A～F 中的某个值，就说明车票的座位是由用户自行选择的。

参数 key_check_isChange 无法确定，要找出该参数的来源，首先分析这个请求是否由单击图 19-25 的“确认”按钮所触发，而图 19-25 的信息核对窗口是单击图 19-18 的“提交订单”按钮所产生的，在这两个过程里面，网页没有发生刷新，新增的请求信息如图 19-26 所示。这就说明，参数 key_check_isChange 可能来自于图 19-18 全部请求信息中的某个请求。因此，我们分别从 XHR、JS 和 Doc 标签查找参数，通过快捷查找，在 Doc 标签中找出参数 key_check_isChange，如图 19-27 所示。

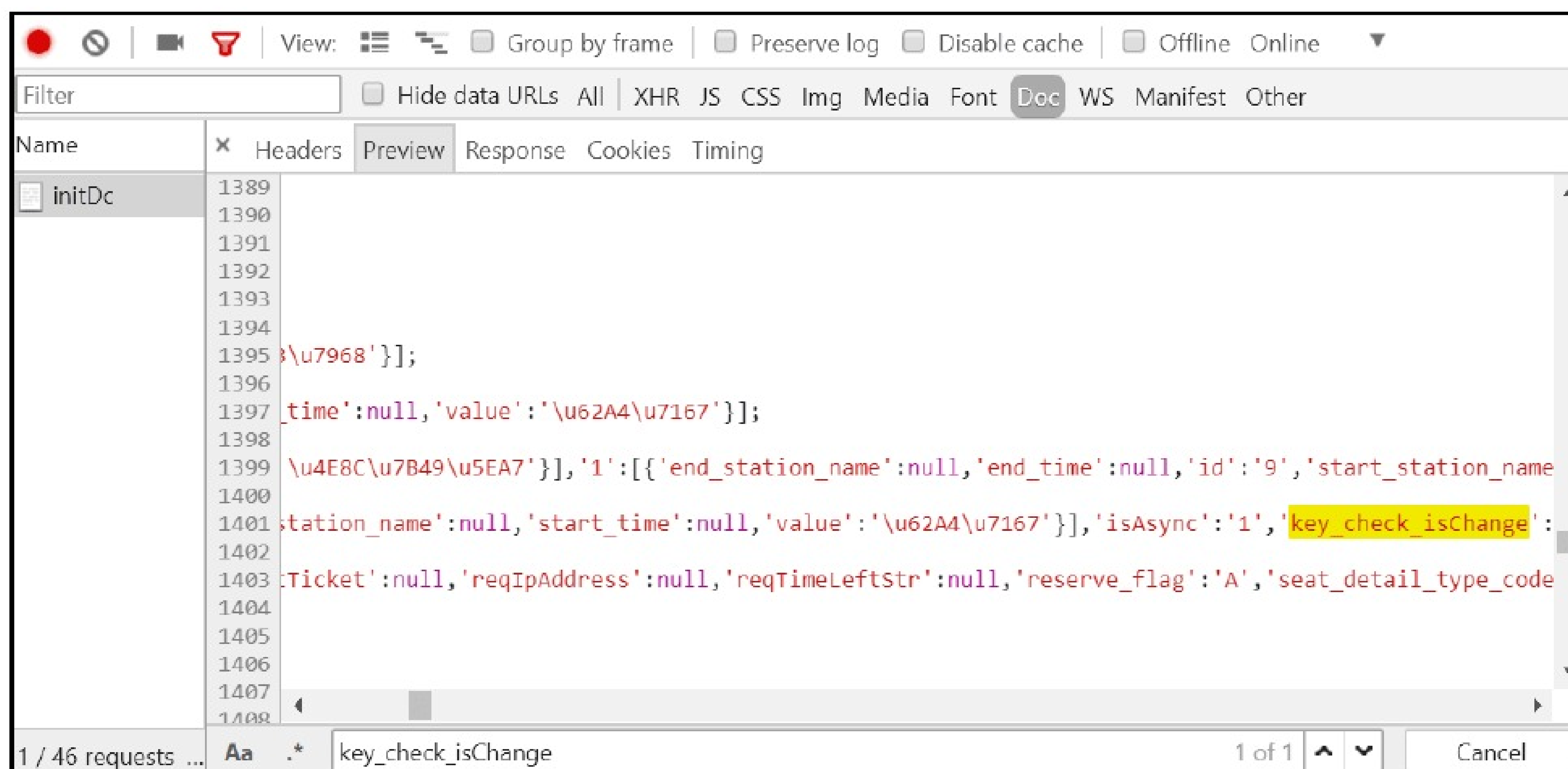


图 19-27　查找请求参数

根据上述分析，订单生成代码如下：

```
url = 'https://kyfw.12306.cn/otn/confirmPassenger/confirmSingleForQueue'
data = {
'passengerTicketStr': passengerTicketStr,
'oldPassengerStr': oldPassengerStr,
'randCode': '',
'purpose codes': '00',
'key check isChange': key check isChange,
'leftTicketStr': train info dict['leftTicket'],
'train_location': train_info_dict['train_location'],
'choose seats': '',
'seatDetailType': '000',
'roomType': '00',
'dwAll': 'N',
' json att': '',
'REPEAT SUBMIT TOKEN': get token
}
```

```
    r = session.post(url,data=data)
    print(r.text)
```

由于此功能与 19.6 节的关联较多，因此在此不再定义新的函数，将此功能直接添加到 19.6 节的函数 creat_order()中，代码如下：

```
    def creat order(name, identity card, phone number, train date,
train info dict):
        # 获取 Doc 标签的数据
        url = 'https://kyfw.12306.cn/otn/confirmPassenger/initDc'
        data = {
            ' json att': ''
        }
        r = session.post(url, data=data)
        # 获取参数
        key check isChange = r.text.split('key check isChange')[1].
                        split(',')[0]. replace(':', '').replace("'", '').strip()
        get_token = r.text.split('globalRepeatSubmitToken')[1].
                        split(';')[0]. replace('=', '').replace("'", '').strip()
        seat code str = r.text.split('ticket seat codeMap=')[1].
                        split(';')[0]. strip()
        # 找出席别编号并去重
        temp_list = re.findall(r"'id':'(.+?)',", seat_code_str)
        temp list = list(set(temp list))
        seatType = temp list[1]

        # 检查订单信息
        # 构建请求参数，name-乘客姓名，identity_card-身份证号
        # phone_number-电话号码，票种为成人票
        oldPassengerStr = name + ',1,' + identity card + ',1 '
        passengerTicketStr = seatType + ',0,1,' + name + ',1,' + identity card
                                 + ',' + phone number + ',N'
        url = 'https://kyfw.12306.cn/otn/confirmPassenger/checkOrderInfo'
        data = {
            'cancel flag': '2',
            'bed_level_order_num': '000000000000000000000000000000',
            'passengerTicketStr': passengerTicketStr,
            'oldPassengerStr': oldPassengerStr,
            'tour flag': 'dc',
            'randCode': '',
            ' json att': '',
            'REPEAT SUBMIT TOKEN': get token
        }
        r = session.post(url, data=data)
        # 提交订单信息
        # train date，train no,stationTrainCode
        # fromStationTelecode, toStationTelecode
        # leftTicket、train location 来自车次信息
        # seatType 和 REPEAT_SUBMIT_TOKEN 来自 Doc 标签的数据
        # purpose_codes 和 _json_att 固定不变
        while 1:
            url = 'https://kyfw.12306.cn/otn/confirmPassenger/getQueueCount'
            #日期格式化处理
            check_ticket_date = train_date + ' 00:00:00'
```

```
        timeArray = time.strptime(check ticket date, "%Y-%m-%d %H:%M:%S")
        date = time.strftime("%a %b %d %Y", timeArray)
        data = {
            'train_date': date + ' GMT+0800 (中国标准时间)',
            'train no': train info dict['train no'],
            'stationTrainCode': train info dict['stationTrainCode'],
            'seatType': seatType,
            'fromStationTelecode': train info dict['fromStationTelecode'],
            'toStationTelecode': train info dict['toStationTelecode'],
            #leftTicket 进行数据格式化处理
            'leftTicket': parse.unquote(train info dict['leftTicket']),
            'purpose codes': '00',
            'train location': train info dict['train location'],
            ' json att': '',
            'REPEAT SUBMIT TOKEN': get token
        }
        r = session.post(url, data=data)
        print(r.text)
        # 判断请求是否成功
        if '系统繁忙，请稍后重试' not in str(r.text):
            break
    # 生成订单
    url = 'https://kyfw.12306.cn/otn/confirmPassenger/confirmSingleForQueue'
    data = {
        'passengerTicketStr': passengerTicketStr,
        'oldPassengerStr': oldPassengerStr,
        'randCode': '',
        'purpose codes': '00',
        'key check isChange': key check isChange,
        'leftTicketStr': train info dict['leftTicket'],
        'train location': train info dict['train location'],
        'choose seats': '',
        'seatDetailType': '000',
        'roomType': '00',
        'dwAll': 'N',
        '_json_att': '',
        'REPEAT SUBMIT TOKEN': get token
    }
    r = session.post(url, data=data)
    print(r.text)
```

19.8 本 章 小 结

本章介绍了 12306 抢票爬虫的编写技巧，整个项目的要点总结如下：

1. 项目实现的爬虫功能

（1）验证码验证。

（2）用户登录与验证。

（3）查询车票。

（4）预订车票。

（5）提交订单。

（6）生成订单。

2. 5 个函数的功能和使用

- login()：用户验证和登录，将验证码验证和用户登录与验证合并在该函数中。
- city_name()：获取城市的编号，将城市名称转换为城市的英文编号。
- train_info()：查询车次，并调用 city_name()。
- train_order()：预订车票，主要生成订单信息。
- creat_order()：填写订单信息并提交确认，将提交订单和生成订单功能合并在该函数中。

3. 项目整体代码

```
import requests
import time
import datetime
import re
from urllib import parse

# 用户登录
def login(username, password):
    #坐标参考：40,40,114,35,192,39,257,36,42,115,119,107,185,124,272,117
    code_list = {
        '1': '40,40,',
        '2': '114,35,',
        '3': '192,39,',
        '4': '257,36,',
        '5': '42,115,',
        '6': '119,107,',
        '7': '185,124,',
        '8': '272,117'
    }
    #请求头
    headers = { 'User-Agent':
              'Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 '
              '(KHTML, like Gecko) Chrome/63.0.3218.0 Safari/537.36',
              'Referer':
              'https://kyfw.12306.cn/otn/login/init'}

    url = 'https://kyfw.12306.cn/passport/captcha/captcha-image?
            login_site=E&module=login&rand=sjrand'
    #忽略证书验证
    r = session.get(url, headers=headers, verify=False)
    #下载验证码图片
    f = open('code.png','wb')
    f.write(r.content)
    f.close()
    #输入验证码图片位置,多个验证码用英文逗号分开
    code=input("请输入验证码：")
    get_code = ''
    for i in code.split(','):
        # 根据输入每组图片的组号，获取对应的坐标位置
```

```
        get_code += code_list[i]
    #验证码校验
    data={
        'answer':get_code,
        'login_site':'E',
        'rand':'sjrand'
    }
    url = 'https://kyfw.12306.cn/passport/captcha/captcha-check'
    r = session.post(url, data=data)
    print(r.text)
    if '验证码校验失败' not in str(r.text):
        # 用户登录
        url = 'https://kyfw.12306.cn/passport/web/login'
        data = {
            'username': username,
            'password': password,
            'appid': 'otn'
        }
        r = session.post(url, data=data)
        print(r.text)
        if '密码输入错误' not in str(r.text):
            #登录验证第一次请求
            url = 'https://kyfw.12306.cn/passport/web/auth/uamtk'
            data = {
                'appid': 'otn'
            }
            r = session.post(url, data=data)
            #登录验证第二次请求
            newapptk = r.json()['newapptk']
            url = 'https://kyfw.12306.cn/otn/uamauthclient'
            data = {
                'tk': newapptk
            }
            r=session.post(url, data=data)
            print(r.text)
            return True
        else:
            return False
    return False

# 获取城市编号
def city_name():
    url = 'https://kyfw.12306.cn/otn/resources/js/framework
              /station_name.js?station_version=1.9031'
    city_code = session.get(url)
    city_code_list = city_code.text.split("|")
    city_dict = {}
    for k, i in enumerate(city_code_list):
        if '@' in i:
            # 城市名作为字典的键，城市英文编号作为字典的值
            city_dict[city_code_list[k + 1]] = city_code_list[k + 2]
    return (city_dict)

# 获取车次信息
```

```
    def train info(train date,query from station name,query to station name):
        #调用函数 city_name 获取城市编号
        city dict = city name()
        from_station = city_dict[query_from_station_name]
        to station = city dict[query to station name]
        # 获取车次信息
        while 1:
            # 第一次请求
            url = 'https://kyfw.12306.cn/otn/leftTicket/log?
                    leftTicketDTO.train date=%s&leftTicketDTO.from station=
                    %s&leftTicketDTO.to station=%s&purpose codes=ADULT'
                    % (train date, from station, to station)
            r = session.get(url)
            # 第二次请求
            # 请求地址的 query 可能变为 queryA，可通过 try……except 控制
            try:
                url = 'https://kyfw.12306.cn/otn/leftTicket/query?
                        leftTicketDTO.train date=%s&leftTicketDTO.from station=
                        %s&leftTicketDTO.to station=%s&purpose codes=ADULT'
                        % (train date, from station, to station)
                r = session.get(url)
                test = r.json()['data']['result']
            except:
                url = 'https://kyfw.12306.cn/otn/leftTicket/queryA?
                        leftTicketDTO.train date=%s&leftTicketDTO.from station=
                        %s&leftTicketDTO.to station=%s&purpose codes=ADULT'
                        % (train date, from station, to station)
                r = session.get(url)
            time.sleep(2)
            if '非法请求' not in str(r.text) and '"result":[]' not in str(r.text):
                train info info = r.json()
                train info dict = {}
                for i in train_info_info['data']['result']:
                    train info status = i.split('|')
                    if train info status[0] != '':
                        train info dict['secretStr']=train info status[0]
                        train info dict['train no']=train info status[2]
                        train info dict['stationTrainCode']=train info status[3]
                        train info dict['fromStationTelecode']=
                                                            train info status[
4]

                        train info dict['toStationTelecode']=train info status[7]
                        train info dict['leftTicket']=train info status[12]
                        train info dict['train location']=train info status[15]
                        return train info dict

    # 预订车票
    def train order(secretStr, train date, query from station name,
query to station name):
        # 获取当前日期
        back train date = datetime.datetime.now().strftime('%Y-%m-%d')
        # 用户登录检查
        url = 'https://kyfw.12306.cn/otn/login/checkUser'
        data = {
```

```
            ' json att': ''
        }
        r = session.post(url, data=data)
        # 提交车票预订请求
        url = 'https://kyfw.12306.cn/otn/leftTicket/submitOrderRequest'
        data = {
            'secretStr': secretStr,
            'train date': train date,
            'back train date': back train date,
            'tour flag': 'dc',
            'purpose codes': 'ADULT',
            'query_from_station_name': query_from_station_name,
            'query to station name': query to station name,
            'undefined': ''
        }
        r = session.post(url, data=data)

    # 生成订单
    def creat order(name, identity card, phone number, train date,
train_info_dict):
        # 获取 Doc 标签的数据
        url = 'https://kyfw.12306.cn/otn/confirmPassenger/initDc'
        data = {
            ' json att': ''
        }
        r = session.post(url, data=data)
        # 获取参数
        key check isChange = r.text.split('key check isChange')[1].
                             split(',')[0].replace(':', '').
                             replace("'", '').strip()
        get token = r.text.split('globalRepeatSubmitToken')[1].
                    split(';')[0].replace('=', '').
                    replace("'", '').strip()
        seat code str = r.text.split('ticket seat codeMap=')[1].
                        split(';')[0].strip()
        # 找出席别编号并去重
        temp list = re.findall(r"'id':'(.+?)',", seat code str)
        temp list = list(set(temp list))
        seatType = temp list[1]

        # 检查订单信息
        # 构建请求参数，name-乘客姓名，identity_card-身份证号
        # phone number-电话号码，票种为成人票
        oldPassengerStr = name + ',1,' + identity card + ',1 '
        passengerTicketStr = seatType + ',0,1,' + name + ',1,' +
                             identity_card + ',' + phone_number + ',N'
        url = 'https://kyfw.12306.cn/otn/confirmPassenger/checkOrderInfo'
        data = {
            'cancel_flag': '2',
            'bed level order num': '000000000000000000000000000000',
            'passengerTicketStr': passengerTicketStr,
            'oldPassengerStr': oldPassengerStr,
            'tour flag': 'dc',
            'randCode': '',
```

```
            ' json att': '',
            'REPEAT SUBMIT TOKEN': get token
        }
        r = session.post(url, data=data)
        # 提交订单信息
        # leftTicket,train location 来自车次信息
        # seatType 和 REPEAT_SUBMIT_TOKEN 来自 Doc 标签的数据
        # purpose codes 和 json att 固定不变
        while 1:
            url = 'https://kyfw.12306.cn/otn/confirmPassenger/getQueueCount'
            # 日期格式化处理
            check ticket date = train date + ' 00:00:00'
            timeArray = time.strptime(check ticket date, "%Y-%m-%d %H:%M:%S")
            date = time.strftime("%a %b %d %Y", timeArray)
            data = {
                'train date': date + ' GMT+0800 (中国标准时间)',
                'train no': train info dict['train no'],
                'stationTrainCode': train info dict['stationTrainCode'],
                'seatType': seatType,
                'fromStationTelecode': train info dict['fromStationTelecode'],
                'toStationTelecode': train info dict['toStationTelecode'],
                #leftTicket 进行数据格式化处理
                'leftTicket': parse.unquote(train info dict['leftTicket']),
                'purpose codes': '00',
                'train location': train info dict['train location'],
                ' json att': '',
                'REPEAT_SUBMIT_TOKEN': get_token
            }
            r = session.post(url, data=data)
            print(r.text)
            # 判断请求是否成功
            if '系统繁忙，请稍后重试' not in str(r.text):
                break
        # 生成订单
        url = 'https://kyfw.12306.cn/otn/confirmPassenger/confirmSingleForQueue'
        data = {
            'passengerTicketStr': passengerTicketStr,
            'oldPassengerStr': oldPassengerStr,
            'randCode': '',
            'purpose codes': '00',
            'key check isChange': key check isChange,
            'leftTicketStr': train info dict['leftTicket'],
            'train_location': train_info_dict['train_location'],
            'choose seats': '',
            'seatDetailType': '000',
            'roomType': '00',
            'dwAll': 'N',
            ' json att': '',
            'REPEAT SUBMIT TOKEN': get token
        }
        r = session.post(url, data=data)
        print(r.text)

    if __name__ == '__main__':
```

```
    session = requests.session()
    # 网站账号密码
    username = '13435423143'
    password = 'XXXXXXXX'
    login_info = login(username, password)
    if login_info:
        train_date = 'YYYY-MM-DD'
        query_from_station_name = '广州'
        query_to_station_name = '武汉'
        train_info_dict = train_info(train_date,
                          query_from_station_name,query_to_station_name)
        secretStr = parse.unquote(train_info_dict['secretStr'])
        train_order(secretStr, train_date,
                    query_from_station_name, query_to_station_name)
        # 乘客信息
        name = '黄永祥'
        identity_card = 'XXXXXXXXX'
        phone_number = '13435423143'
        creat_order(name, identity_card, phone_number,
                    train_date, train_info_dict)
```

上述代码运行结果如图 19-28 所示。

```
请输入验证码: 4,7
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
{"result_message":"验证码校验成功","result_code":"4"}
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
{"result_message":"登录成功","result_code":0,"uamtk":"iEqRuWXqR9xwI6Qo9vIkU4hE3CN6CeV0WfpgmxhEf3Aaf1110"}
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
{"apptk":"Uzaa7ugBhWHgPTXJuSZ_WrilyNkI-6yT3WXNw840Snk361110","result_code":0,"result_message":"验证通过","username":"黄永祥"}
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
D:\Python\lib\site-packages\urllib3\connectionpool.py:857: InsecureRequestWarning: Unverified HTTPS request is being made. Adding certificate verification is strongly
  InsecureRequestWarning)
{"validateMessagesShowId":"_validatorMessage","status":true,"httpstatus":200,"data":{"count":"0","ticket":"101","op_2":"false","countT":"0","op_1":"false"},"messages"
{"validateMessagesShowId":"_validatorMessage","status":true,"httpstatus":200,"data":{"submitStatus":true},"messages":[],"validateMessages":{}}
```

图 19-28 程序运行结果

从图 19-28 最后的数据看到"httpstatus":200,"data":{"submitStatus":true}，代表购票已成功。用户可以在“我的 12306→未完成订单”中查看已生成的订单内容，最后的付款流程需要用户自行完成，此时整个购票流程真正完成。

4. 进一步完善的建议

本项目只是简单地实现了一个抢票过程，但遇到春运期间的抢票，程序的稳定性需要进一步修改和完善，下面列出几条值得完善的建议：

（1）增加车次可选择功能，将查询出来的车次的发车时间、时长等信息提供给用户自行选择。

（2）判断订单生成状态，对生成失败的订单进行相应处理。

（3）异常处理机制，因为网站的稳定性一直是饱受争议的问题，所以要完善异常处理机制，确保出现异常的时候能及时处理，提高程序的稳定性。

第 20 章

实战：玩转微博

20.1 项 目 分 析

接触过微博的读者都知道，一些热门的微博有很多转发、评论和点赞，而且博主有很庞大的粉丝数。这么高的数据量其实都离不开营销手段，在庞大的数据中有多少是真实数据不为人知，但可以肯定的是，这些数据肯定有水分存在。那么这些有水分的数据是如何产生的呢？这就是本章讲述的重点。

本章主要实现的功能如下。

- weibo_login.py：微博用户登录，同时也是程序运行文件。
- weibo_verify_code.py：第三方平台 API，实现验证码识别。
- weibo_collect.py：根据关键字搜索并采集热门微博。
- weibo_send.py：发布微博。
- weibo_follow.py：关注用户。
- weibo_forward.py：微博点赞和转发评论。
- data.csv：存储采集数据。
- 文件夹 video 和 image：分别存储采集的视频和图片。

20.2 用 户 登 录

进入微博首页，我们发现微博大部分功能都要用户登录才能使用。那么爬取微博的第一步就是实现用户登录。在 Chrome 浏览器对微博的登录机制进行分析，在浏览器中输入 http://weibo.com/，打开开发者工具，捕捉首页的请求信息，如图 20-1 所示。

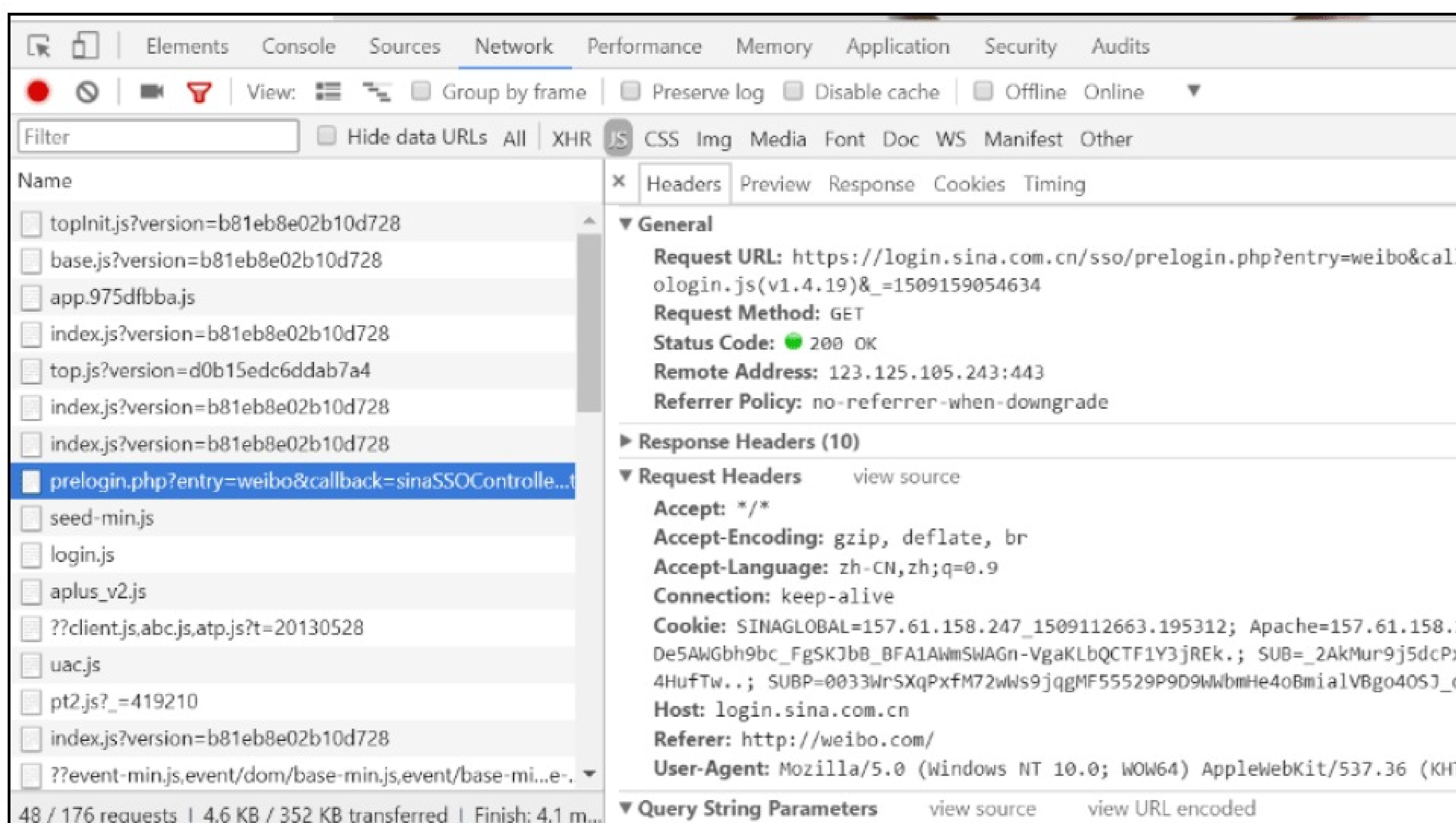


图 20-1　微博登录界面

在开发者工具里分别查看 XHR、JS 和 Doc 标签的请求信息：

（1）Doc 标签有 4 个请求信息，请求信息的响应内容都是 HTML，主要是网页的布局和一些 JavaScript 脚本信息。

（2）XHR 标签有一个 POST 的请求信息，对该信息的请求链接、请求参数和响应内容进行分析，该请求信息与登录信息没有太大关联。

（3）JS 标签有多个请求信息，大多数请求都是 JavaScript 脚本内容，查看每一个请求信息，发现其中一个请求较为特殊，如图 20-2 所示。

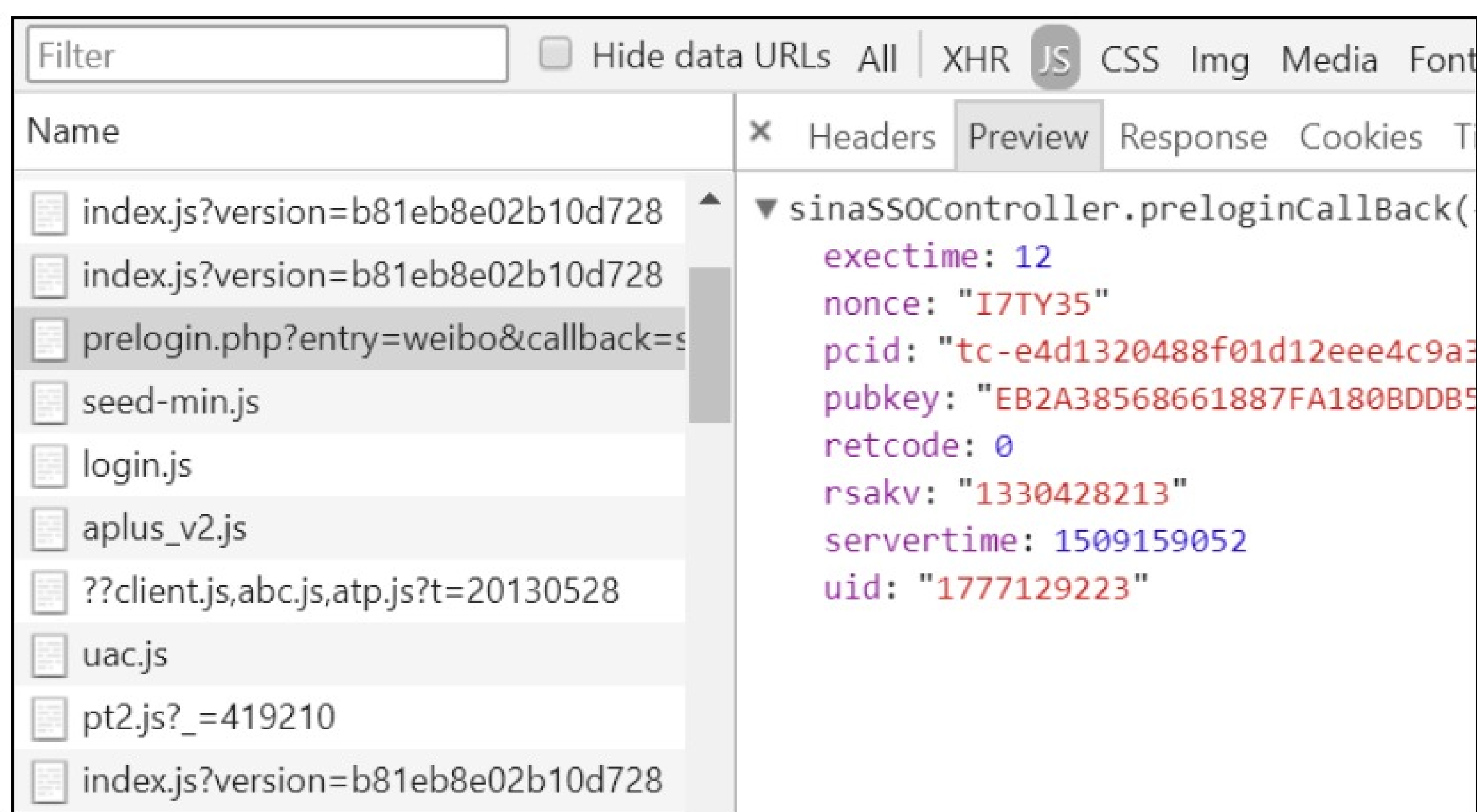


图 20-2　微博登录分析

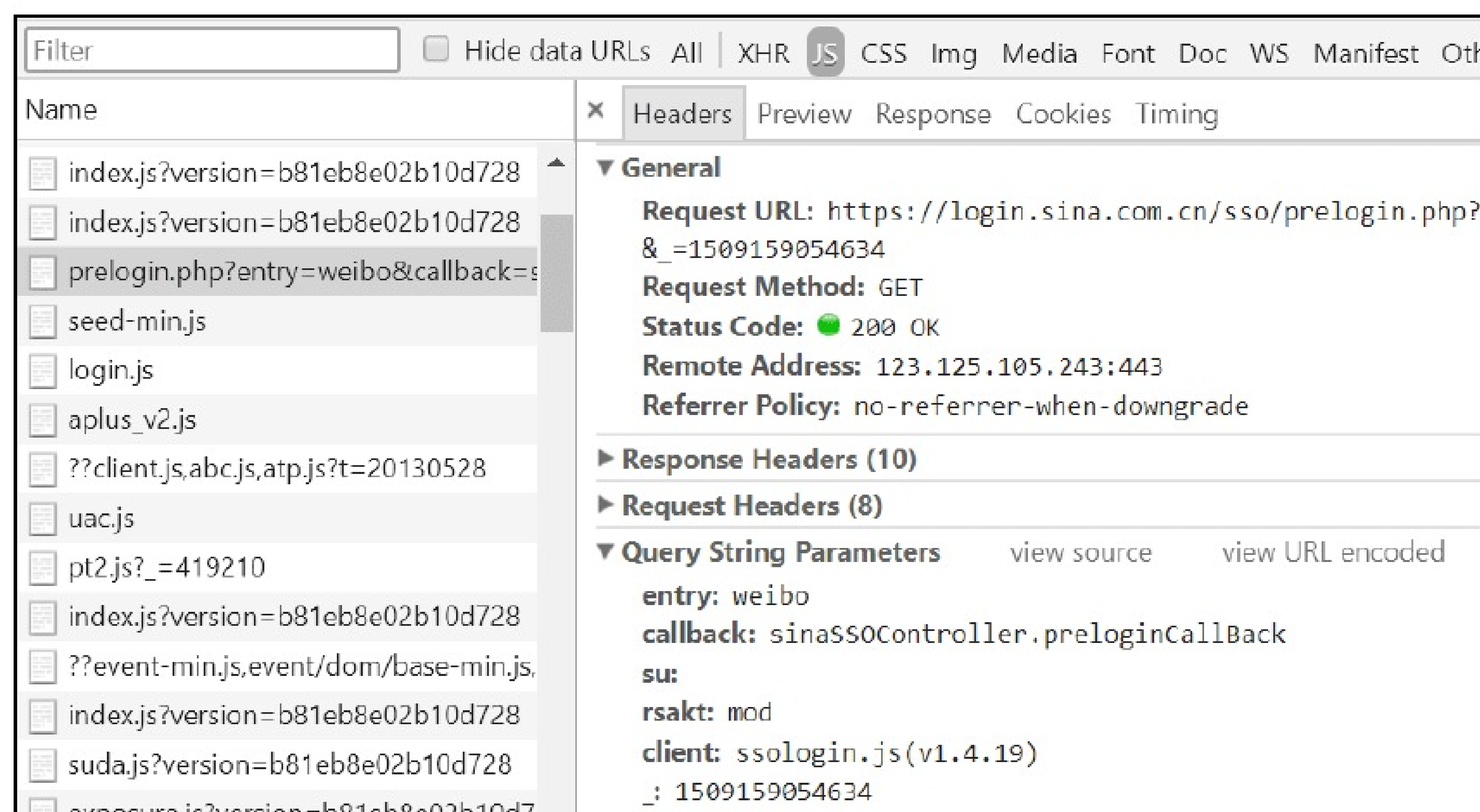


图 20-2（续）

根据图 20-2 分析请求参数和响应内容，含义如下。

- 请求参数 su：代表用户账号，一般以 su 或 username 命名。
- 请求参数 1509159054634：以“150”开头的数字大多数是时间戳。
- 请求参数 rsakt 和响应内容 rsakv：无法确定这两个参数代表的含义，但两者都含有 rsa，rsa 是一个加密方法。参数值可能经过加密处理。
- 响应内容 pubkey：中文翻译为公共密钥，从这个参数可知，某些数据肯定做过加密处理，大多数是对账号、密码做加密处理。

通过简单分析，我们知道在用户登录之前会触发一个准备登录（prelogin）请求，该请求中包含一些加密信息。也就是说，在实现登录功能之前，先要对上述请求信息发送请求，获取其响应内容的加密信息后，才能进行下一步用户登录。实现代码如下：

```
import requests
import time
def get server data(su):
    # 构建 URL
    prelogin url = 'https://login.sina.com.cn/sso/prelogin.php?
                    entry=weibo&callback=sinaSSOController.prelogin
                    CallBack&su=%s&rsakt=mod&client=ssologin.js(v1.
                    4.19)& =%s' %(su, str(int(time.time() * 1000)))
    pre data res = session.get(prelogin url, headers=headers ,
                   proxies=proxies)
    # 将响应内容转换为字典格式
    sever data = eval(pre data res.content.decode("utf-8").
                replace("sinaSSOController.preloginCallBack", ''))
    return sever data
if   name   == "  main  ":
    # 构造请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
             Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent
    }
```

```
    # 代理 IP，防止同一 IP 登录多个不同微博账号
    proxies = {}
    # 新建会话
    session = requests.session()
    # 用户账号
    su = '13435423143'
    sever_data = get_server_data(su)
```

现在得到了登录的加密信息，但还不知道具体使用了哪些加密方法，我们知道网站对数据加密一般都在前端完成加密处理，然后将加密的数据发送到网站后台，在后台再对数据解密处理并返回响应，这样的方法可以提高数据在发送传输时的安全性。

根据上述原理，我们可以在请求信息中找出具体的加密方法，数据加密主要以 JavaScript 实现，对 JS 标签里的各个 JS 文件进行分析，找到实现加密功能的 JS 文件，如图 20-3 所示。

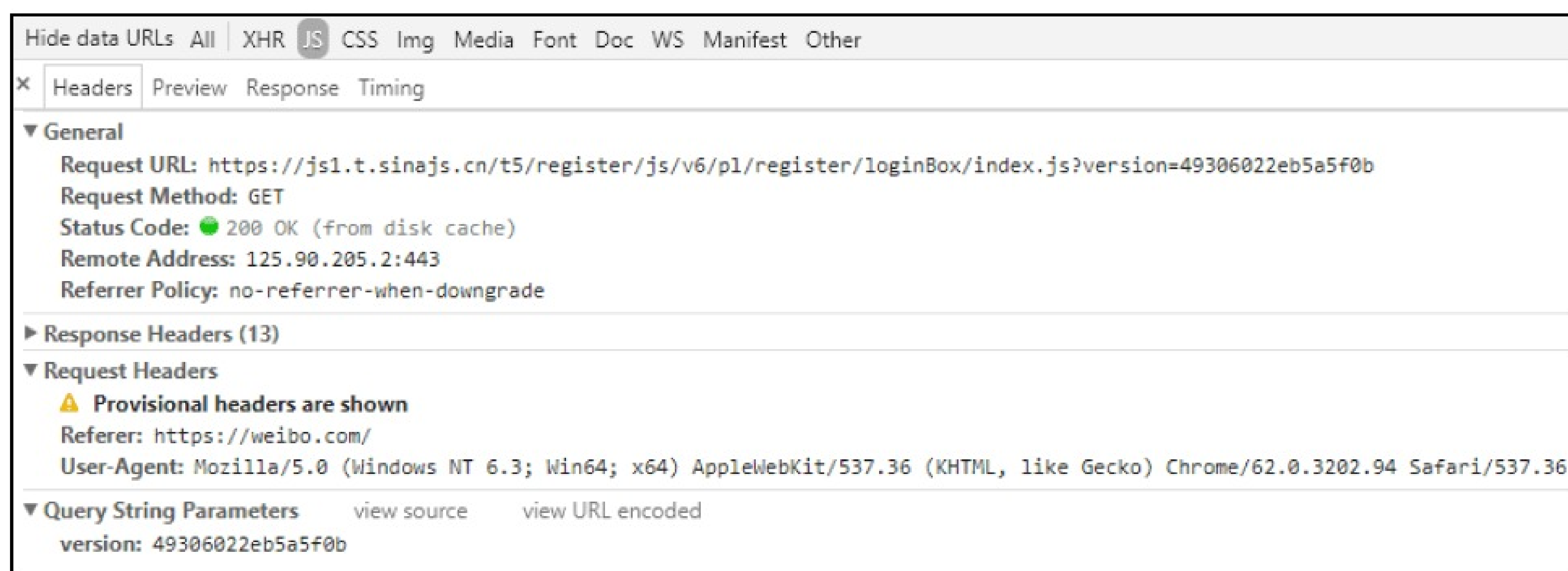


图 20-3　微博登录加密方法

通过分析图 20-3 中请求信息的响应内容（JavaScript 代码）可以发现：

（1）用户账号主要使用 base64 方式加密。

（2）密码是使用 RSA 加密的，加密密钥是图 20-2 中的 servertime、nonce 和 pubkey。

根据上述分析，我们得知账号和密码使用了不同的加密方式，对此分别对两者定义不同的函数，代码如下：

```
import urllib
import base64
import rsa
import binascii
# 账号加密
def get_su(username):
    # 使用 urllib.parse.quote_plus 对 email 地址或手机号码的特殊符号编码处理
    # 然后使用 base64 加密
    username_quote = urllib.parse.quote_plus(username)
    username_base64 = base64.b64encode(username_quote.encode("utf-8"))
    return username_base64.decode("utf-8")

# 密码加密，servertime、nonce、pubkey 是来自图 16-2 的数据
def get_password(password, servertime, nonce, pubkey):
    rsaPublickey = int(pubkey, 16)
    # 创建公钥
```

```
        key = rsa.PublicKey(rsaPublickey, 65537)
        # 拼接明文
        message = str(servertime) + '\t' + str(nonce) + '\n' + str(password)
        message = message.encode("utf-8")
        # 加密
        passwd = rsa.encrypt(message, key)
        # 将加密信息转换为16进制
        passwd = binascii.b2a_hex(passwd)
        return passwd
```

注　意
rsa 模块是第三方库，可使用 pip install rsa 安装。

完成用户的账号、密码加密处理后，最后一步就是实现用户登录，在浏览器中输入账号、密码，单击“登录”按钮，分析开发者工具捕捉到的请求信息，如图 20-4 所示。

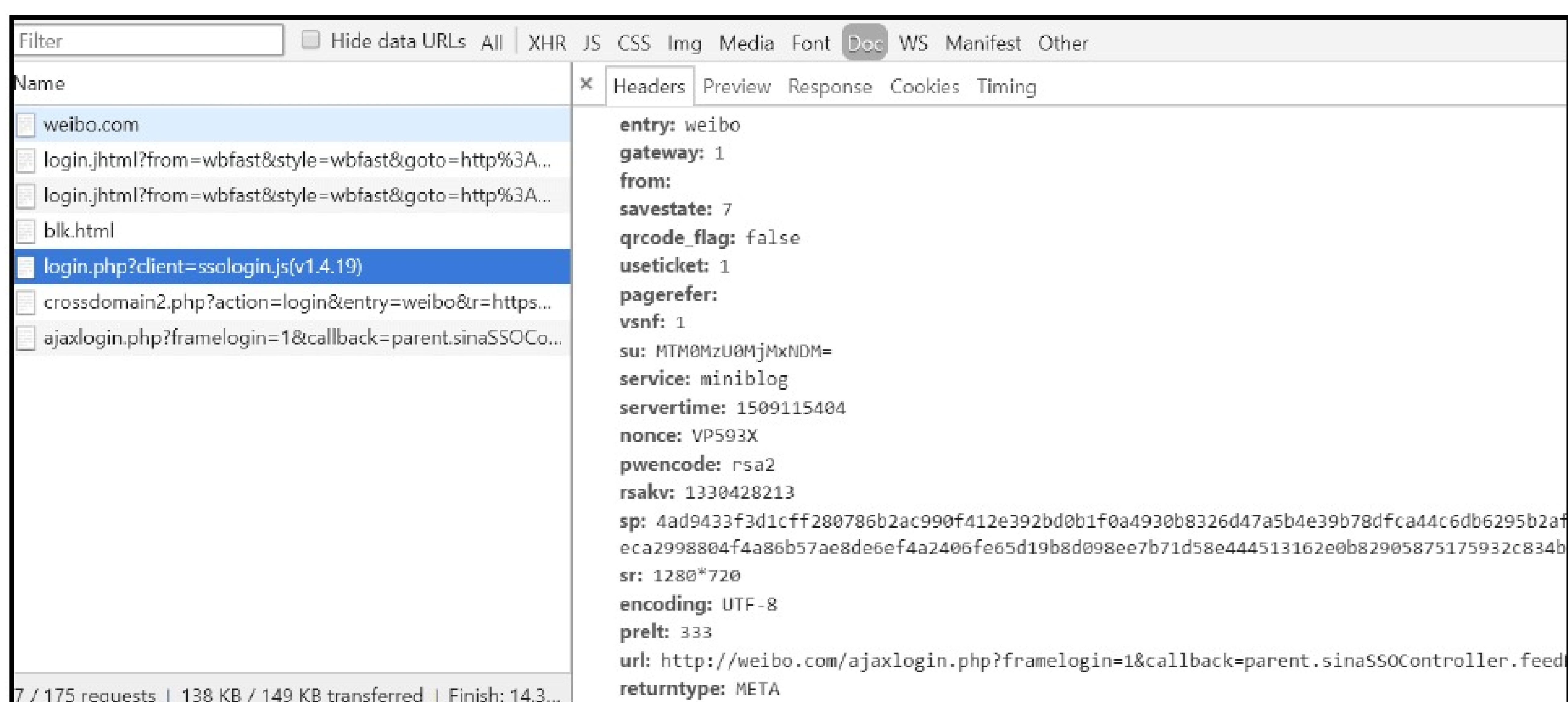


图 20-4　微博用户登录

从请求参数可知，su、sp、servertime、nonce 和 rsakv 是动态变化的，其他参数都是固定不变的。而 servertime、nonce 和 rsakv 可以在图 20-2 中直接获取，su 和 sp 分别是加密后的账号和密码。用户登录代码实现如下：

```
import time
import base64
import rsa
import binascii
import requests
import re,urllib
def login(username, password):
    # 获取servertime、nonce、rsakv、su和sp
    su = get_su(username)
    sever_data = get_server_data(su)
    servertime = sever_data["servertime"]
    nonce = sever_data['nonce']
    rsakv = sever_data["rsakv"]
    pubkey = sever_data["pubkey"]
    sp = get_password(password, servertime, nonce, pubkey)
    # 构建请求参数
```

```
    data = {
        'entry': 'weibo',
        'gateway': '1',
        'from': '',
        'savestate': '7',
        'useticket': '1',
        'pagerefer': "http://login.sina.com.cn/sso/logout.php?entry=
                 miniblog&r=http%3A%2F%2Fweibo.com%2Flogout.php%3Fbackurl",
        'vsnf': '1',
        'su': su,
        'service': 'miniblog',
        'servertime': servertime,
        'nonce': nonce,
        'pwencode': 'rsa2',
        'rsakv': rsakv,
        'sp': sp,
        'sr': '1366*768',
        'encoding': 'UTF-8',
        'prelt': '115',
        'url': 'http://weibo.com/ajaxlogin.php?framelogin=1&
                  callback= parent. sinaSSOController.feedBackUrlCallBack',
        'returntype': 'META'
        }
    #用户登录
    url = 'http://login.sina.com.cn/sso/login.php?
            client=ssologin. js(v1.4.18)'
    login page = session.post(url, data=data,proxies=proxies)
    print(login page.text) ①
if   name   == "  main  ":
    # 请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
              Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent
    }
    # 代理 IP，防止同一 IP 登录多个不同微博账号
    proxies = {}
    # 新建会话
    session = requests.session()
    login('13435423143','xxxxxxxxxx')
```

运行上述代码，结果如图 20-5 所示。



图 20-5　用户登录响应内容一

响应内容是一个 HTML 格式的数据，在 HTML 内容中无法得知是否登录成功，因为在数据中无法获取用户的信息。但细心分析可知，HTML 内容中有“location.replace”，这是一个页面跳转的功能，以此作为突破口，可以尝试访问跳转的链接，看能否在这个链接中获取用户信息。在上述代码中的①处添加以下代码：

```
login_loop = (login_page.content.decode("GBK"))
# 网页跳转 URL，获取用户信息
pa = r'location\.replace\([\'"](.*?)[\'"]\)'
loop_url = re.findall(pa, login_loop)[0]
login_index = session.get(loop_url, proxies=proxies)
print(login_index.text) ②
```

再次运行代码，结果如图 20-6 和图 20-7 所示。

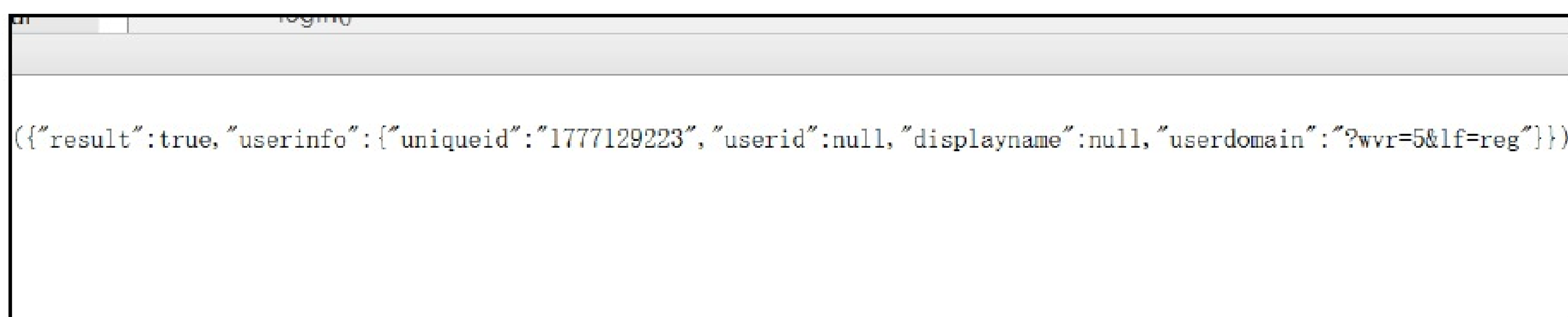


图 20-6　用户登录响应内容二

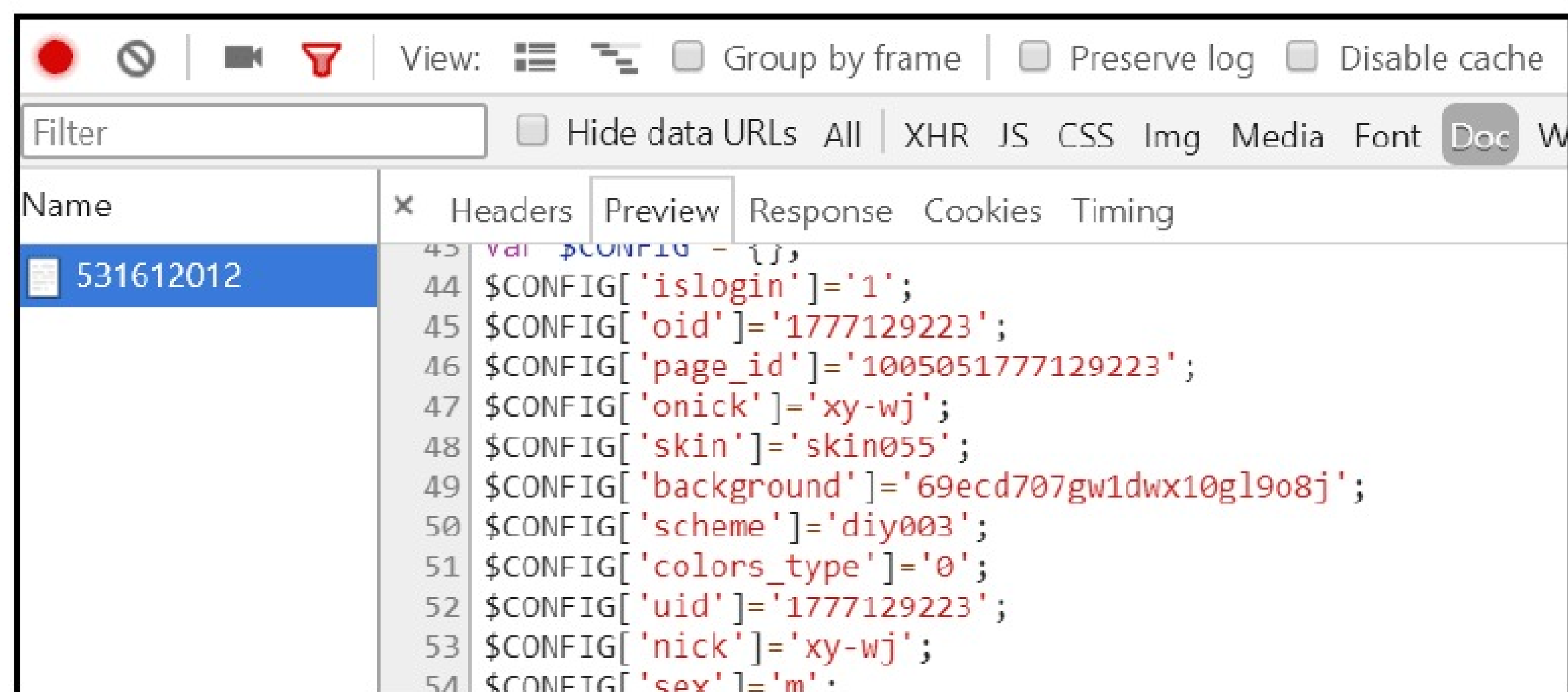


图 20-7　微博用户信息

图 20-7 是在网页上查看的微博用户首页信息。对比图 20-6 和图 20-7，图 20-6 说明用户已成功登录，其中 userinfo 代表用户信息，观察 userinfo 的数据，发现 uniqueid 等于图 20-7 中的 uid 和 oid，因此根据 uniqueid 获取用户首页信息，在上述代码的②处加入以下代码：

```
uuid = login index.text
uuid pa = r'"uniqueid":"(.*?)"'
uuid res = re.findall(uuid pa, uuid, re.S)[0]
# 根据 uniqueid 构建微博首页的 URL
web weibo url = "http://weibo.com/%s" % uuid res
weibo page = session.get(web weibo url, proxies=proxies)
response = weibo page.text
person info = {}
if '$CONFIG' in response:
    person info['nick'] = response.split("$CONFIG['nick']='")
                          [1].split("';")[0]
    person info['watermark']=response.split("$CONFIG['watermark']= '")
                             [1].split("';")[0]
    person info['location'] = response.split("$CONFIG['location']= '")
                             [1].split("';")[0]
    person info['uid']=response.split("$CONFIG['uid']='")[1].split("';")[0]
    person info['domain'] = response.split("$CONFIG['domain']= '")
                           [1].split("';")[0]
    person info['oid'] = response.split("$CONFIG['oid']='")
                         [1].split("';")[0]
    print('登录成功，你的用户名为：' + person_info['nick'])
```

综合上述已实现的功能，本节完整的代码如下：

```
import requests
import time
import urllib
import base64
import rsa
import binascii
import re
# 登录前准备
def get server data(su):
    # 构建 URL
    prelogin url = 'https://login.sina.com.cn/sso/prelogin.php?
                    entry= weibo&callback=sinaSSOController.
                    preloginCallBack&su=%s&rsakt=mod&client=
                   ssologin.js(v1.4.19)& =%s' %(su, str(int(
                    time.time() * 1000)))
    pre data res = session.get(prelogin url, headers=headers,
                   proxies=proxies)
    # 将响应内容转换为字典格式
    sever data = eval(pre data res.content.decode("utf-8").
                replace("sinaSSOController.preloginCallBack", ''))
    return sever_data

# 账号加密
def get su(username):
    # 使用 urllib.parse.quote plus 对 email 地址或手机号码的特殊符号进行编码处理
    # 然后使用 base64 加密
```

```
    username_quote = urllib.parse.quote_plus(username)
    username_base64 = base64.b64encode(username_quote.encode("utf-8"))
    return username_base64.decode("utf-8")

# 密码加密，servertime、nonce、pubkey 是来自图 16-2 的数据
def get_password(password, servertime, nonce, pubkey):
    rsaPublickey = int(pubkey, 16)
    # 创建公钥
    key = rsa.PublicKey(rsaPublickey, 65537)
    # 拼接明文
    message = str(servertime) + '\t' + str(nonce) + '\n' + str(password)
    message = message.encode("utf-8")
    # 加密
    passwd = rsa.encrypt(message, key)
    # 将加密信息转换为 16 进制
    passwd = binascii.b2a_hex(passwd)
    return passwd

# 用户登录
def login(username, password):
    # 获取 servertime、nonce、rsakv、su 和 sp
    su = get_su(username)
    sever_data = get_server_data(su)
    servertime = sever_data["servertime"]
    nonce = sever_data['nonce']
    rsakv = sever_data["rsakv"]
    pubkey = sever_data["pubkey"]
    sp = get_password(password, servertime, nonce, pubkey)
    # 构建请求参数
    data = {
        'entry': 'weibo',
        'gateway': '1',
        'from': '',
        'savestate': '7',
        'useticket': '1',
        'pagerefer': "http://login.sina.com.cn/sso/logout.php?entry=
              miniblog&r = http%3A%2F%2Fweibo.com%2Flogout.php%3Fbackurl",
        'vsnf': '1',
        'su': su,
        'service': 'miniblog',
        'servertime': servertime,
        'nonce': nonce,
        'pwencode': 'rsa2',
        'rsakv': rsakv,
        'sp': sp,
        'sr': '1366*768',
        'encoding': 'UTF-8',
        'prelt': '115',
        'url': 'http://weibo.com/ajaxlogin.php?framelogin=1&callback=
               parent.sinaSSOController.feedBackUrlCallBack',
        'returntype': 'META'
        }
    # 用户登录
    login_url = 'http://login.sina.com.cn/sso/login.php?
```

```
                 client= ssologin.js(v1.4.18)'
    login_page = session.post(login_url, data=data)
    login_loop = (login_page.content.decode("GBK"))
    # 网页跳转 URL，获取用户信息
    pa = r'location\.replace\([\'"](.*?)[\'"]\)'
    loop_url = re.findall(pa, login_loop)[0]
    login_index = session.get(loop_url)
    uuid = login_index.text
    uuid_pa = r'"uniqueid":"(.*?)"'
    uuid_res = re.findall(uuid_pa, uuid, re.S)[0]
    # 根据 uniqueid 构建微博首页 URL
    web_weibo_url = "http://weibo.com/%s" % uuid_res
    weibo_page = session.get(web_weibo_url)
    response = weibo_page.text
    person_info = {}
    if '$CONFIG' in response:
        person_info['nick'] = response.split("$CONFIG['nick']= '")
                              [1].split("';")[0]
        person_info['watermark'] = response.split("$CONFIG['watermark']= '")
                              [1].split("';")[0]
        person_info['location'] = response.split("$CONFIG['location']= '")
                              [1].split("';")[0]
        person_info['uid'] = response.split("$CONFIG['uid']= '")
                              [1].split("';")[0]
        person_info['domain'] = response.split("$CONFIG['domain']= '")
                              [1].split("';")[0]
        person_info['oid'] = response.split("$CONFIG['oid']= '")
                              [1].split("';")[0]
        print('登录成功，你的用户名为：' + person_info['nick'])
    else:
        print('登录失败')
    return person_info

if __name__ == "__main__":
    # 构造请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
             Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent
    }
    # 代理 IP
    proxies = {}
    # 新建会话
    session = requests.session()
    user_info = login('13435423143','xxxxxx')
```

20.3　用户登录（带验证码）

20.2 节已实现微博用户登录，如果要实现多账号批量登录，那么需要使用代理 IP 实现，否则同一个 IP 登录多个账号，账号很容易被网站查封。在使用代理 IP 登录微博时，有可能遇到验证码

验证的问题，为解决验证码验证问题，我们在浏览器上设置代理 IP，如图 20-8 所示。

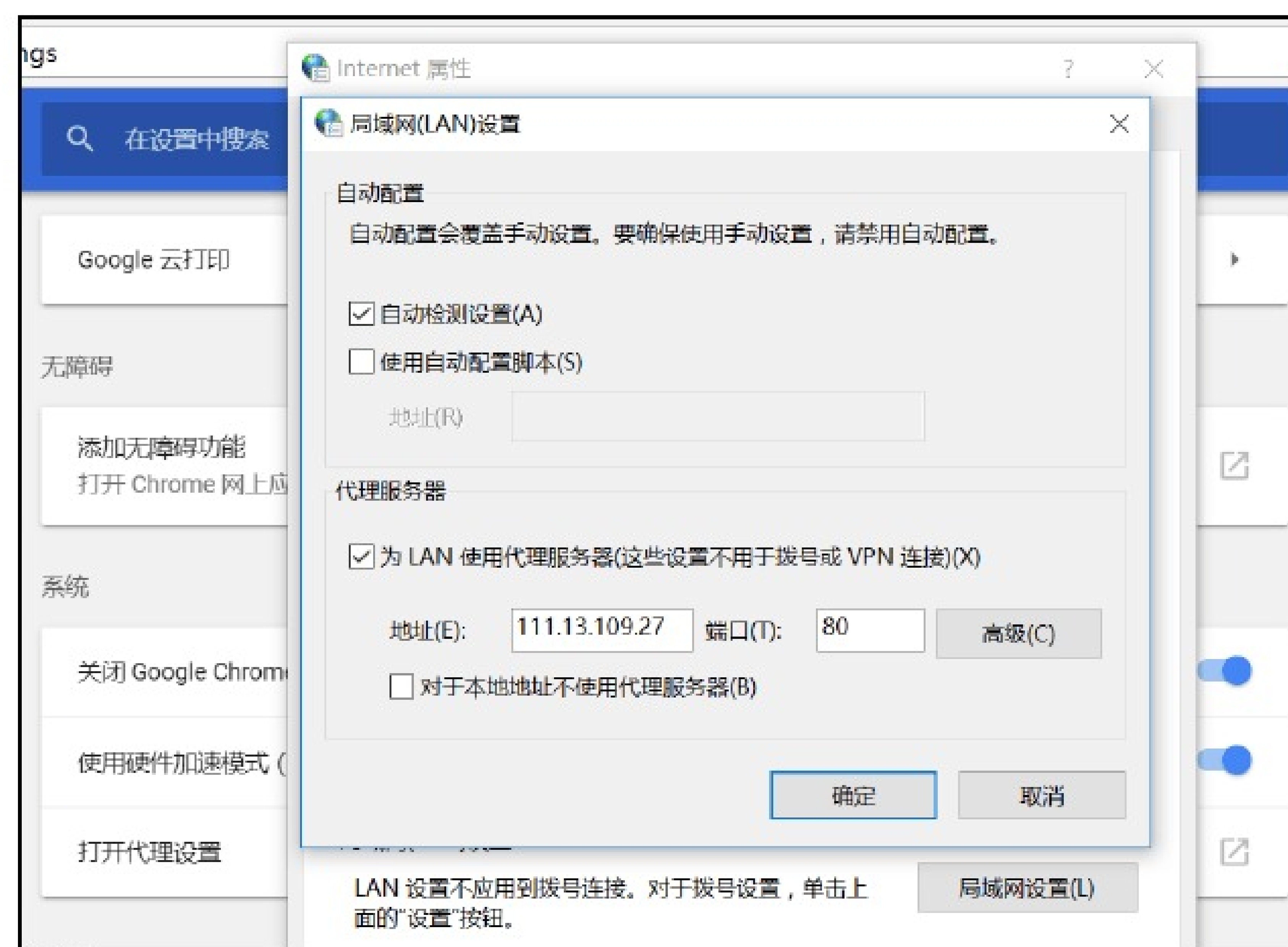


图 20-8 设置代理 IP

设置代理 IP 之后，返回微博登录界面，可以看到登录界面出现验证码，如图 20-9 所示。



图 20-9 带验证码微博登录

打开开发者工具，查看请求信息进行分析，找到验证码图片请求信息，如图 20-10 所示。

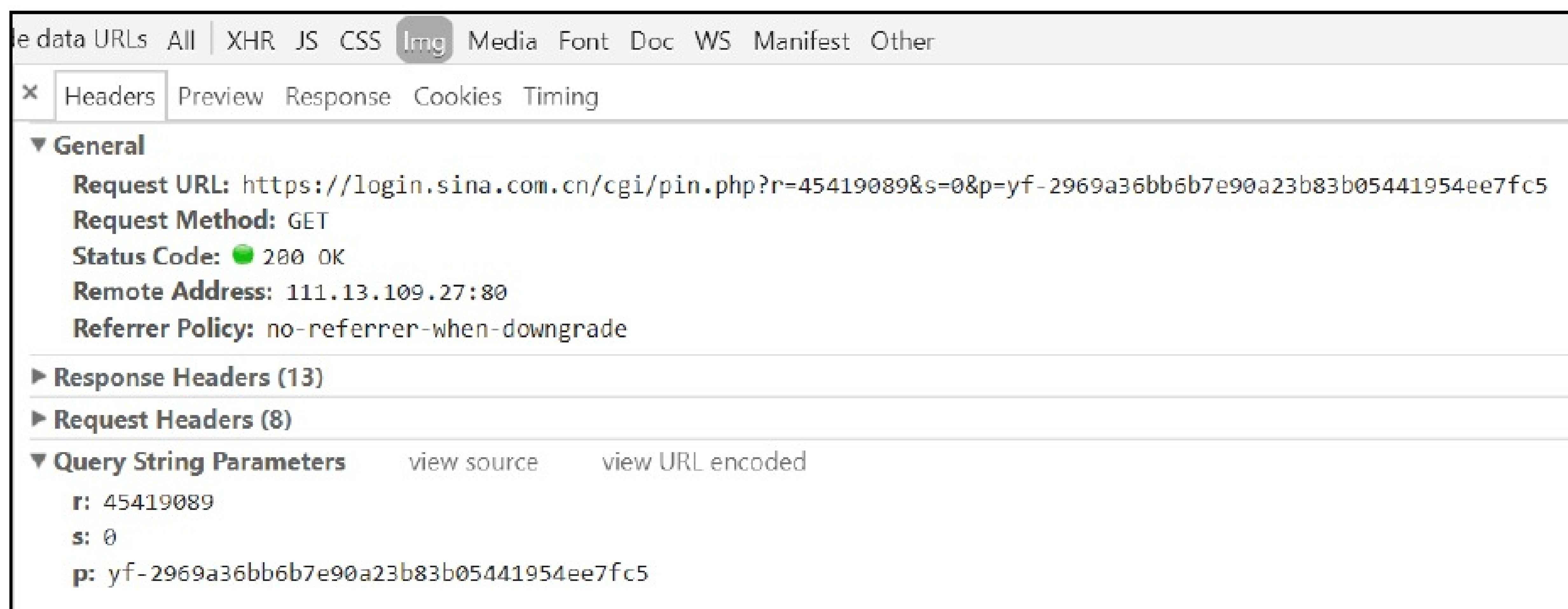


图 20-10 图片验证码请求信息

从图 20-10 中看到有三个请求参数：r 是一个随机数，生成的规律不固定；s 是一个固定数字；p 是一个不可知的数据，需要找出该数据的来源。

再分析登录前的加密信息（prelogin）是否也发生变化，如图 20-11 所示。

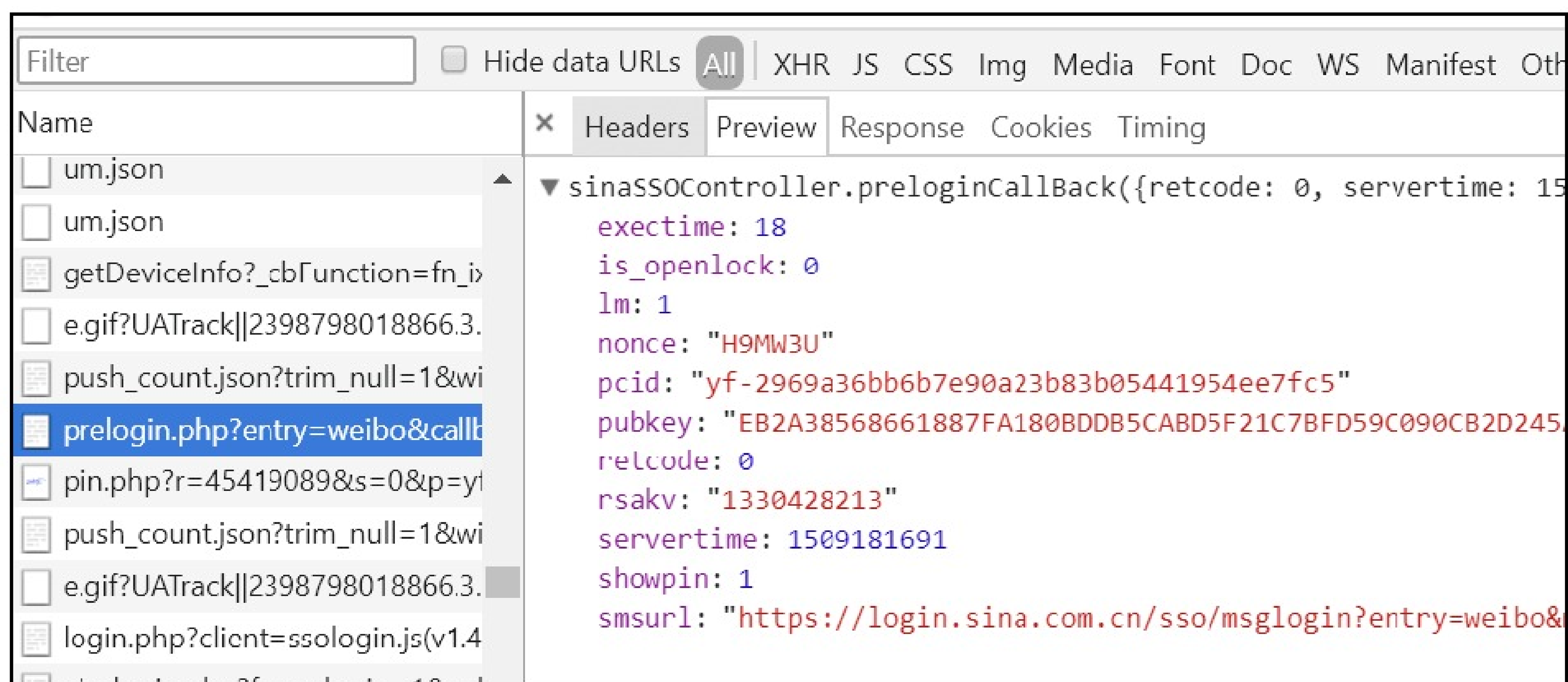


图 20-11　带验证码的登录信息

对比图 20-11 与图 20-2，发现带验证码的登录前加密信息中的数据多了 showpin、is_openlock 和 lm；再与图 20-10 对比，发现图 20-11 中 pcid 的数据与图 20-10 中 p 参数的值相同。

结合上述分析：

（1）访问加密信息，根据返回内容进行判断，如果存在 showpin、is_openlock 和 lm 数据，就说明当前登录需要验证码识别。

（2）如果存在验证码，就先下载验证码图片，再进行下一步的用户登录；否则直接执行 20.2 节的代码。

下载验证码图片的代码如下：

```
def get_img(pcid):
    url = 'https://login.sina.com.cn/cgi/pin.php?r=%s&s=0&p=%s'
          %(str(math.floor(random.random() * 100000000)),pcid)
    resp = session.get(url)
    verify_code_path = '%s.png' % (str(int(time.time() * 1000)))
    f = open(verify_code_path, 'wb')
    f.write(resp.content)
    f.close()
    return verify_code_path
```

在函数 get_img()中，参数 pcid 由加密信息的 pcid 传递，最后函数返回的是图片的相对路径。完成验证码下载后，接着分析带验证码的登录请求，如图 20-12 所示。

从图 20-12 和图 20-4 的请求参数对比得出，图 20-12 的请求参数多了 pcid 和 door。pcid 是来自加密信息里的响应数据，door 是验证码图片内容。

根据上述分析，总结如下：

（1）如果带有验证码，加密信息的响应内容就含有 showpin、is_openlock 和 lm 数据。

（2）判断加密信息的响应内容是否需要下载验证码图片。

（3）下载图片验证码后，需要对验证码进行识别。

（4）在用户登录请求中，带验证码的登录需要添加参数 pcid 和 door。

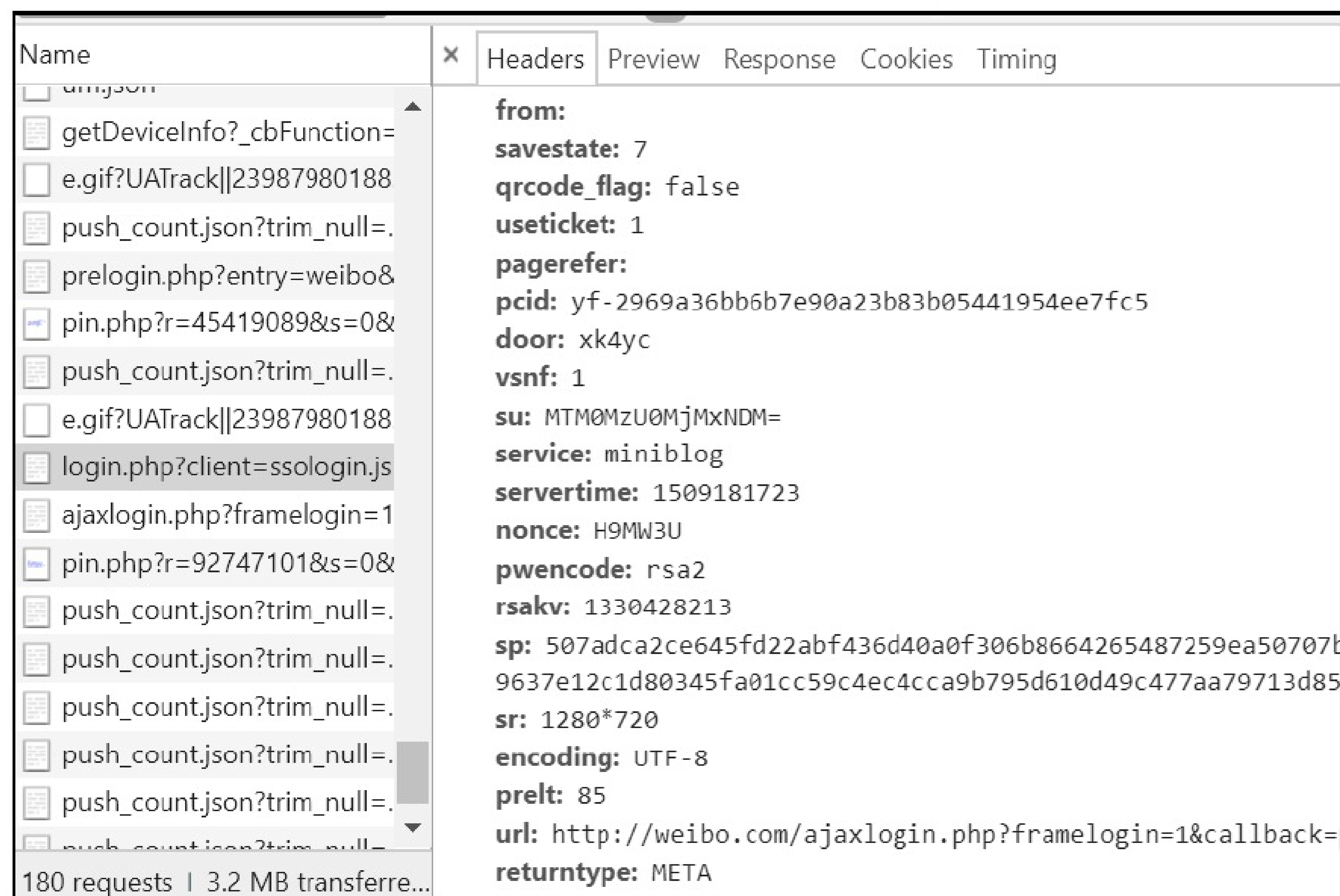


图 20-12　带验证码用户登录

在上面的分析要点中，目前还没解决验证码识别的问题。12.3 节讲述了第三方平台如何识别验证码，因此本项目使用第三方平台提供的 API 解决验证码识别问题，将 API 代码命名并保存在文件 weibo_verify_code.py 中。在 20.2 节的代码中加入验证码处理功能，代码如下：

```
import requests
import time
import urllib
import base64
import rsa
import binascii
import re
# 接入第三方 API 识别验证码
from weibo verify code import code verificate
# 登录前准备
def get server data(su):
    # 构建 URL
    prelogin url = 'https://login.sina.com.cn/sso/prelogin.php?entry=
                    weibo&callback=sinaSSOController.preloginCallBack&
                    su=%s&rsakt=mod&client=ssologin.js(v1.4.19)& =%s'
                    %(su, str(int(time.time() * 1000)))
    pre data res = session.get(prelogin url, headers=headers,
                    proxies=proxies)
    # 将响应内容转换为字典格式
    sever data = eval(pre data res.content.decode("utf-8").
                replace("sinaSSOController.preloginCallBack", ''))
    return sever data

# 账号加密
```

```
def get_su(username):
    # 使用 urllib.parse.quote_plus 对 email 地址或手机号码的特殊符号进行编码处理
    # 然后使用 base64 加密
    username_quote = urllib.parse.quote_plus(username)
    username_base64 = base64.b64encode(username_quote.encode("utf-8"))
    return username_base64.decode("utf-8")

# 密码加密，servertime、nonce、pubkey 是来自图 16-2 的数据
def get_password(password, servertime, nonce, pubkey):
    rsaPublickey = int(pubkey, 16)
    # 创建公钥
    key = rsa.PublicKey(rsaPublickey, 65537)
    # 拼接明文
    message = str(servertime) + '\t' + str(nonce) + '\n' + str(password)
    message = message.encode("utf-8")
    # 加密
    passwd = rsa.encrypt(message, key)
    # 将加密信息转换为 16 进制
    passwd = binascii.b2a_hex(passwd)
    return passwd

# 下载验证码图片
def get_img(pcid):
    url = 'https://login.sina.com.cn/cgi/pin.php?r=%s&s=0&p=%s'
           %(str(math.floor(random.random() * 100000000)),pcid)
    resp = session.get(url)
    verify_code_path = '%s.png' % (str(int(time.time() * 1000)))
    f = open(verify_code_path, 'wb')
    f.write(resp.content)
    f.close()
    return verify_code_path

# 用户登录
def login(username, password):
    # 获取 servertime、nonce、rsakv、su 和 sp
    su = get_su(username)
    sever_data = get_server_data(su)
    servertime = sever_data["servertime"]
    nonce = sever_data['nonce']
    rsakv = sever_data["rsakv"]
    pubkey = sever_data["pubkey"]
    sp = get_password(password, servertime, nonce, pubkey)
    # 构建请求参数
    data = {
        'entry': 'weibo',
        'gateway': '1',
        'from': '',
        'savestate': '7',
        'useticket': '1',
        'pagerefer': "http://login.sina.com.cn/sso/logout.php?entry=
              miniblog&r=http%3A%2F%2Fweibo.com%2Flogout.php%3Fbackurl",
        'vsnf': '1',
        'su': su,
        'service': 'miniblog',
```

```
            'servertime': servertime,
            'nonce': nonce,
            'pwencode': 'rsa2',
            'rsakv': rsakv,
            'sp': sp,
            'sr': '1366*768',
            'encoding': 'UTF-8',
            'prelt': '115',
            'url': 'http://weibo.com/ajaxlogin.php?framelogin=1&callback=
                    parent.sinaSSOController.feedBackUrlCallBack',
            'returntype': 'META'
            }
        # 判断是否存在验证码
        if 'showpin' in sever data.keys():
            # 添加请求参数
            pcid = sever data['pcid']
            data['pcid'] = pcid
            # 下载验证码图片
            verify code path = get img(pcid)
            # 第三方平台识别验证码
            verify code = code verificate(yundama username, yundama password,
                                           verify code path)
            print(verify code)
            data['door'] = verify code

        # 用户登录
        login url = 'http://login.sina.com.cn/sso/login.php?client=
                      ssologin.js(v1.4.18)'
        login page = session.post(login url, data=data)
        login loop = (login page.content.decode("GBK"))
        # 网页跳转 URL，获取用户信息
        pa = r'location\.replace\([\'"](.*?)[\'"]\)'
        loop_url = re.findall(pa, login_loop)[0]
        login index = session.get(loop url)
        uuid = login index.text
        uuid pa = r'"uniqueid":"(.*?)"'
        uuid res = re.findall(uuid pa, uuid, re.S)[0]
        # 根据 uniqueid 构建微博首页 URL
        web weibo url = "http://weibo.com/%s" % uuid res
        weibo page = session.get(web weibo url)
        response = weibo page.text
        person info = {}
        if '$CONFIG' in response:
            person info['nick'] = response.split("$CONFIG['nick']= '")
                                  [1].split("';")[0]
            person_info['watermark'] = response.split("$CONFIG['watermark']= '")
                                  [1].split("';")[0]
            person info['location'] = response.split("$CONFIG['location']= '")
                                  [1].split("';")[0]
            person info['uid'] = response.split("$CONFIG['uid']= '")
                                  [1].split("';")[0]
            person info['domain'] = response.split("$CONFIG['domain']= '")
                                  [1].split("';")[0]
            person_info['oid'] = response.split("$CONFIG['oid']= '")
```

```
                              [1].split("';")[0]
        print('登录成功，你的用户名为：' + person_info['nick'])
    else:
        print('登录失败')
    return person_info

if __name__ == "__main__":
    # 构造请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
             Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent
    }
    # 代理 IP
    proxies = {
        'http': "http://113.214.13.1:8000/"
    }
    # 新建会话
    session = requests.session()
    # 第三方平台账号密码
    yundama_username = 'xxxxxx'
    yundama_password = 'xxxxxx'
    user_info = login('13435423143','xxxxxx')
```

程序运行结果如图 20-13 所示。

```
F:\Python\python.exe F:/微博/weibo_Software/weibo_Software/weibo_login.py
uid: 53927
balance: 1472
cid: 1573126148, result: VMYMK
VMYMK
登陆成功，你的用户名为：xy-wj

Process finished with exit code 0
```

图 20-13　带验证码的微博用户登录结果

20.4　关键词搜索热门微博

完成用户登录后，接着实现关键字搜索热门微博，该功能可以让我们及时掌握微博最新的咨询以及各个行业的动态走向，巧妙运用这个功能等于拥有了微博平台的大数据。

在浏览器中打开网站 https://s.weibo.com/，以“#王者荣耀#”为关键字搜索，如图 20-14 所示。

每次搜索不同的关键词，网页都会重新刷新一遍，说明搜索结果是由网站后台直接生成。由于搜索内容是经过分页处理，因此把页数切换到第二页，然后在 Doc 标签下分析当前的请求信息，如图 20-15 所示。

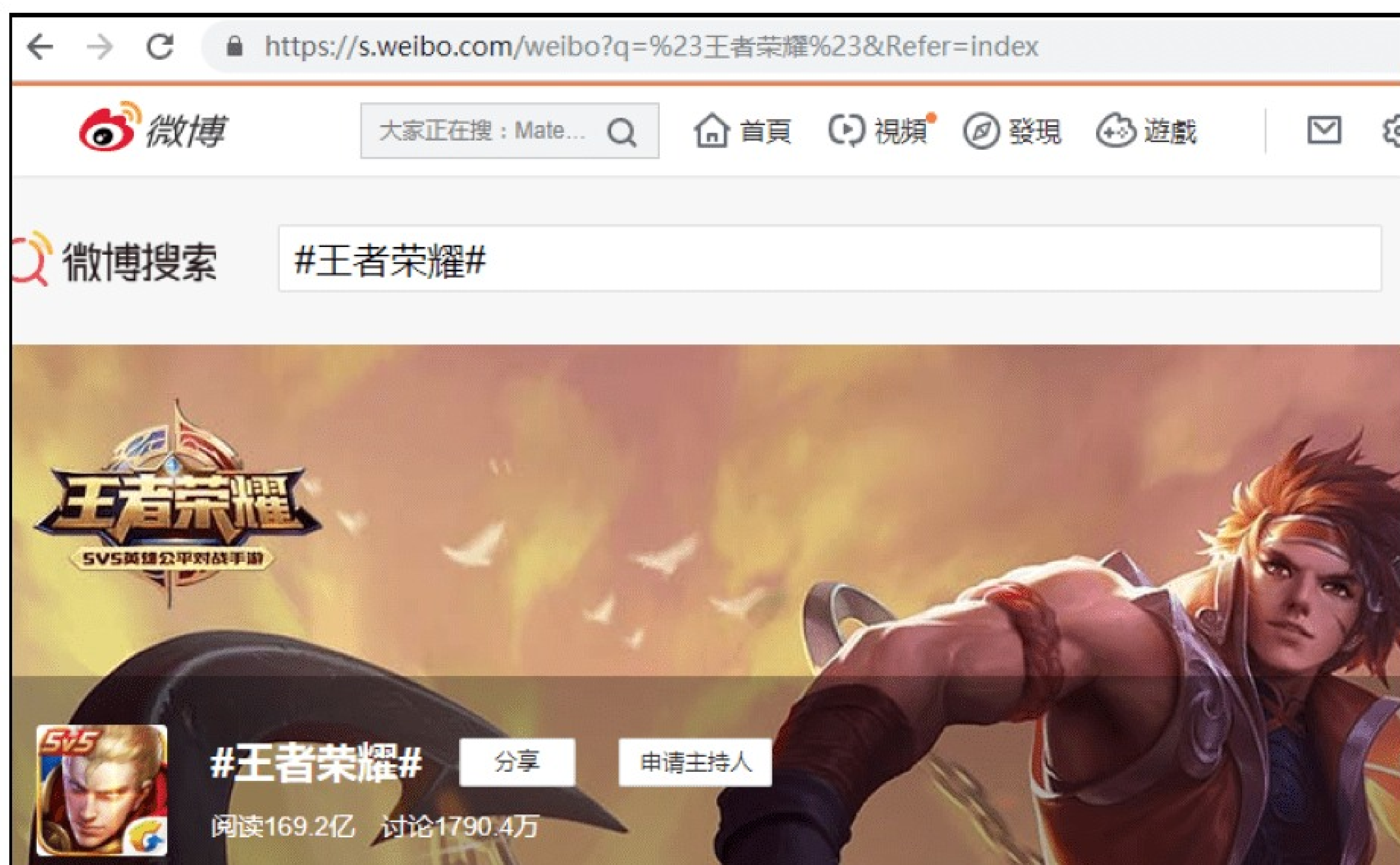


图 20-14　关键词搜索微博



图 20-15　关键词搜索微博的请求信息

根据图 20-15 上的请求信息，该请求的请求参数分析如下：

- URL 含有“%23”等字符，说明 URL 对中文进行了编码处理，编码处理由 urllib.parse.quote()实现。
- Refer 的参数值是固定不变的。
- page 代表页数，一个关键词最多返回 50 页内容。

分析该请求的响应内容，发现每一条微博信息都存放在<div class="content" node-type="like">，在此标签内，可以分别找出微博用户、微博内容、图片文件和视频文件的所在位置，由于网页内容较多，本书只列出部分内容的所在位置，如图 20-16 所示。



图 20-16　微博内容

分析网页的 HTML 结构得知，微博用户、微博内容、图片文件和视频文件的 HTML 结构说明如下：

- 微博用户在<a>标签，属性 class="name"。
- 微博内容在<a>标签，该标签的上级标签是<p>标签，属性 class="txt"。有时候<p>标签下含有两个<a>标签，这是因为微博内容过长，需要单击“展开全文”才能看到完整的内容，而第二个<a>标签就是完整的微博内容。
- 图片文件在<img>标签，该标签的上级标签是<ul>标签，属性 class="m3"。
- 视频文件在<a>标签，属性 class="WB_video_h5"，文件路径在属性 action-data。

综合上述，可以确定采集数据的具体位置，实现代码如下：

```
from bs4 import BeautifulSoup
import urllib
import csv
import requests
import time
import datetime
from concurrent.futures import ThreadPoolExecutor

# 多线程爬取视频文件
def thread_video(get_video_value, video_path):
    if get_video_value:
        # 提取视频文件的 URL 地址
        url = get_video_value['action-data'].
                split('video_src=')[1].split('&cover_img=')[0]
        url = 'http:' + urllib.parse.unquote(url)
        try:
            temp_value = requests.get(url)
            video = open('video/' + video_path, 'wb')
            video.write(temp_value.content)
            video.close()
```

```
        except:
            pass

# 多线程爬取图片
def thread img(k, img path):
    r = requests.get('http:' + k['src'])
    img = open('image/' + img path, 'wb')
    img.write(r.content)
    img.close()

# 采集微博
def collect(keyword, session, pagenumber=1):
    # 关键字编码
    now = datetime.datetime.now().strftime('%Y-%m-%d')
    # 构建 URL
    keyword = urllib.parse.quote(keyword)
    url = 'https://s.weibo.com/weibo?q=' + keyword +
          '&Refer = SWeibo bo&page=%s' % (str(pagenumber))
    r = session.get(url)
    # 清洗多余的符号
    get value = r.text.replace('\/', '/')
    soup = BeautifulSoup(get value, 'html5lib')
    # 定位用户信息
    get info = soup.find all('div', class ="content")

    for i in get info:
        # 微博内容与用户信息
        get comment = i.find all('p', class ='txt')
        if get comment:
            # 输出全部文字内容
            if len(get comment) > 1:
                get comment = get comment[-1]
            else:
                get comment = get comment[0]
            comment = get comment.getText().strip()
            # 获取用户信息
            get user = i.find('a', class ='name')
            if get user:
                user name = get user.getText().strip()
            else:
                user name = ''

        img_path_list = ''
        # 获取图片内容
        get img value = i.find('ul', class ='m3')
        # 输出图片
        if get img value:
            get img value = get img value.find all('img')
            for k in get img value:
                img path = str(int(time.time() * 1000)) + '.jpg'
                img path list = img path list + img path + '/'
                pool = ThreadPoolExecutor(max workers=1)
                pool.submit(thread img, k, img path)
```

```
        video path = ''
        # 输出视频
        get video value = i.find('a', class ='WB video h5')
        if get_video_value:
            pool = ThreadPoolExecutor(max workers=1)
            video path = str(int(time.time() * 1000)) + '.mp4'
            pool.submit(thread video, get video value, video path)

        # 用于生成 csv
        f = open('data.csv', 'a', newline='', encoding='gb18030')
        writer = csv.writer(f)
        writer.writerow([user name,comment,img path list,video path,now])
        f.close()
```

整段代码共由以下三个函数组成。

- thread_img(): 多线程下载图片。
- thread_video(): 多线程下载视频。
- collect(): 实现微博采集，函数参数 keyword、session 和 pagenumber 分别是关键字、带有用户信息的会话对象和采集页数。因为本节的代码存放在文件 weibo_collect.py 中，与用户登录的代码不在同一个文件，所以需要将带有用户信息的会话对象传递给该函数。

函数 collect()实现的功能依次如下：

- 对函数参数 keyword 执行一次 URL 编码。
- 构建请求链接并发送请求，并对响应内容进行清洗处理。
- 采集微博用户信息、文字内容、图片和视频。图片和视频的下载分别调用函数 thread_img() 和 thread_video()，使用多线程下载文件，提高爬取速度。

代码运行需要结合 20.3 节的登录功能一起使用，将本节代码存放在文件 weibo_collect.py 中，并且与文件 weibo_login.py 同一目录，当前目录下必须有文件夹 image 和 video，否则下载的图片和视频无法保存。修改 weibo_login.py 文件，修改的代码如下：

```
if __name__ == "__main__":
    # 构造请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
              Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent
    }
    # 代理 IP
    proxies = {}
    # 新建会话
    session = requests.session()
    # 第三方平台账号、密码
    yundama username = 'xxxx'
    yundama_password = 'xxxx'
    user info = login('13435423143','xxxx')
    # 导入微博采集功能
    from weibo collect import collect
    #爬取前 10 页数据
```

```
    for i in range(10):
        collect('#王者荣耀#',session,i)
```

20.5 发 布 微 博

发布微博是在浏览器上编辑好要发布的内容，然后单击“发布”按钮进行发布。微博中有很多可以编辑的功能，如插入表情、话题、图片、视频和定时发送等。其中，表情和话题可以归纳为文字内容。本节主要实现文字内容、图片和定时发送的微博发布。

在浏览器上分别捕捉三种不同发布方式的请求，如图 20-17~图 20-19 所示。

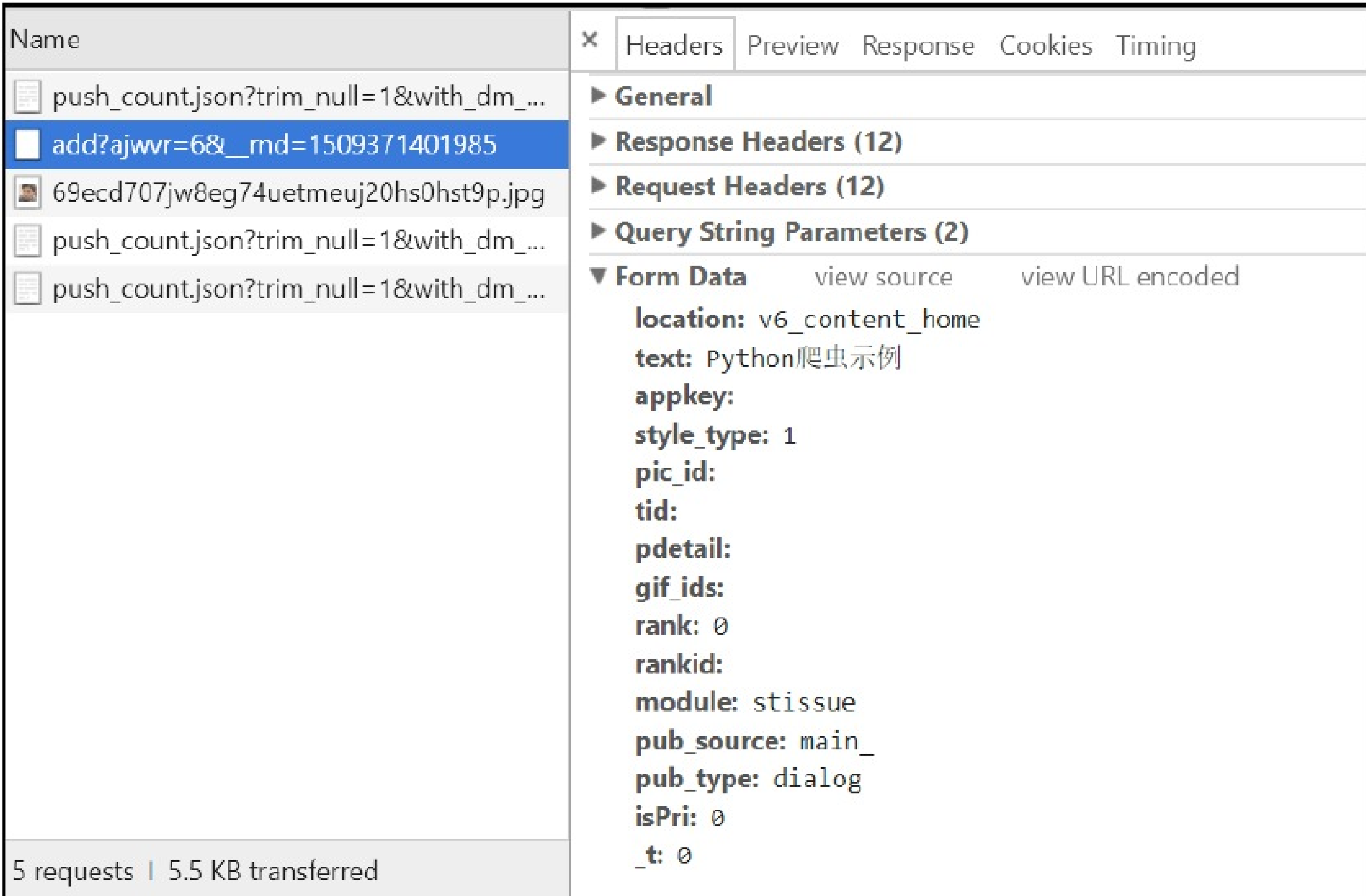


图 20-17　微博发布——文字内容

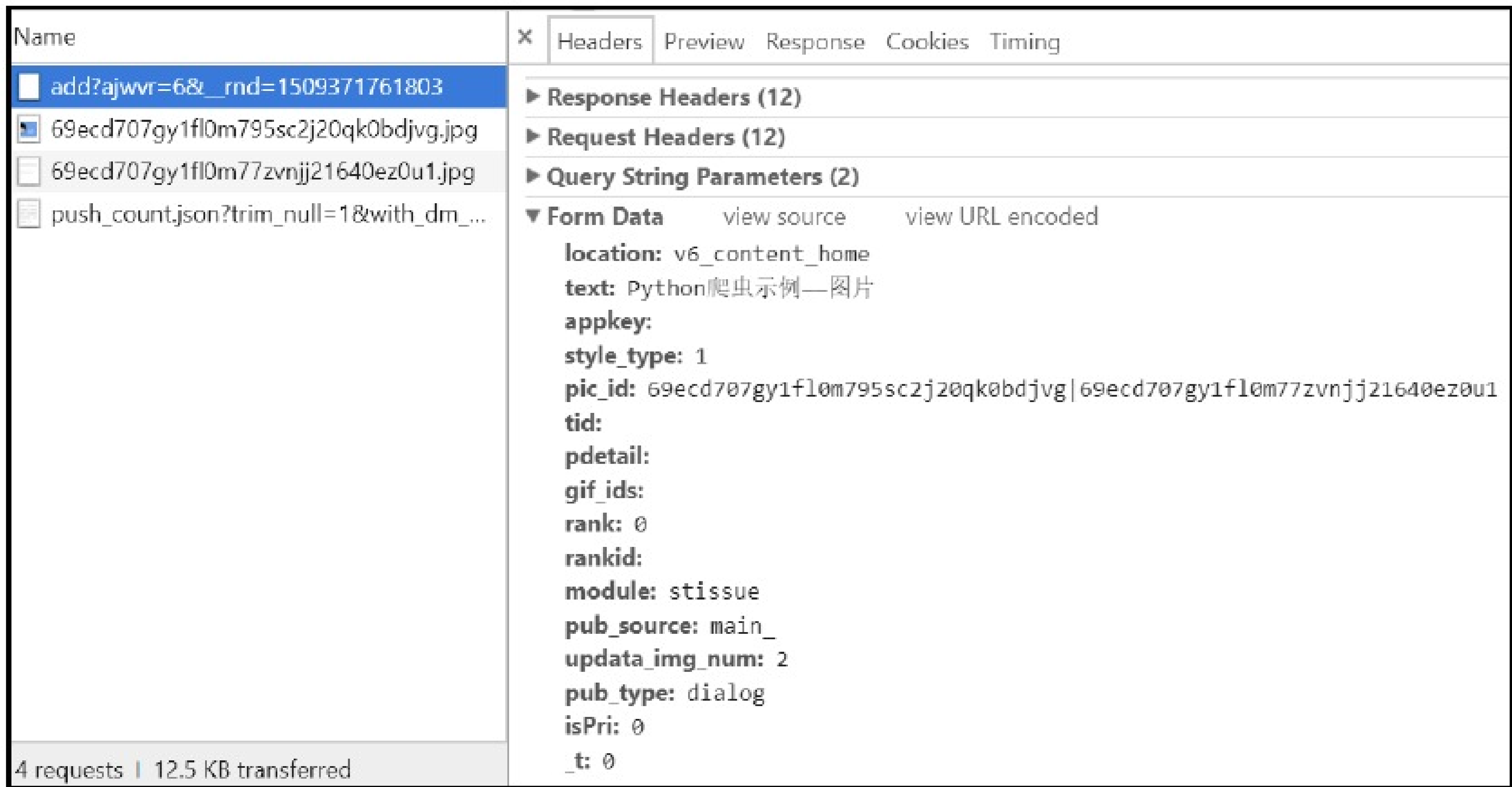


图 20-18　微博发布——文字内容和图片

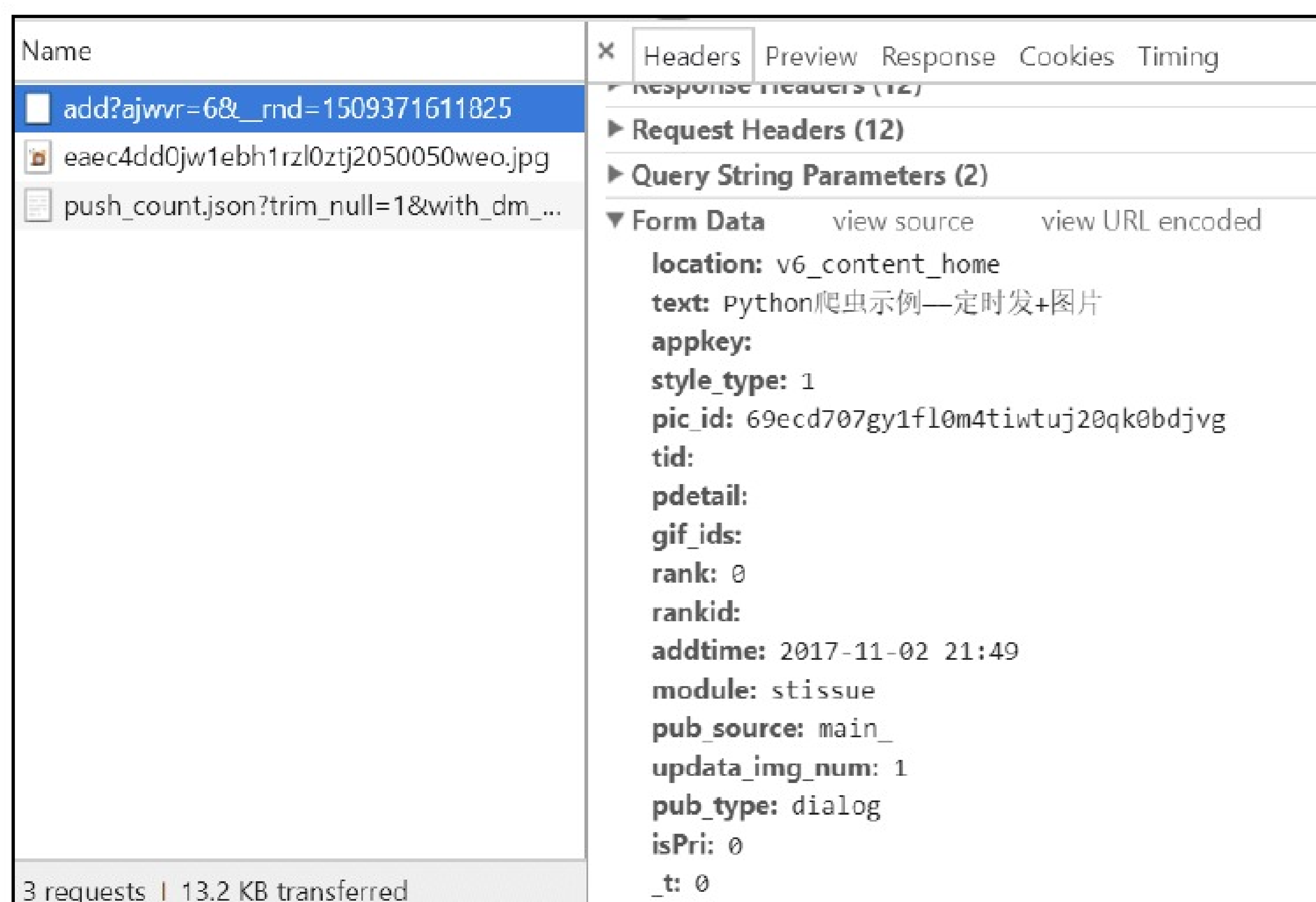


图 20-19　微博定时发布

通过对比三种不同的发布方式的请求可以发现，三者的请求链接一致，唯一的区别在于请求参数的差异。请求参数的差异如下：

（1）对比图 20-18 和图 20-19 的请求参数，图 20-19 多出了参数 updata_img_num，该参数是所发布的图片的数量。参数 pic_id 的值非空，从参数名分析可知，参数 pic_id 应该是图片的 id。

（2）对比图 20-17 和图 20-19 的请求参数，图 20-19 多出了参数 addtime，该参数是发布时间；再对比两者的参数 pic_id 和 updata_img_num，图 20-18 比图 20-19 多一张图片，图 20-18 的参数 pic_id 将每张图片之间用“|”隔开。

（3）参数 location 和 text 分别是用户信息和发布的文字内容，其他参数都是固定不变的。

经过上述分析，现在无法确定 pic_id 的数据来源，该参数如果是图片的 id，那么在添加图片的时候，网站应该会对添加的图片生成一个图片 id，用于标识图片。为了验证猜想，我们捕捉添加图片时所触发的请求信息，如图 20-20 所示。

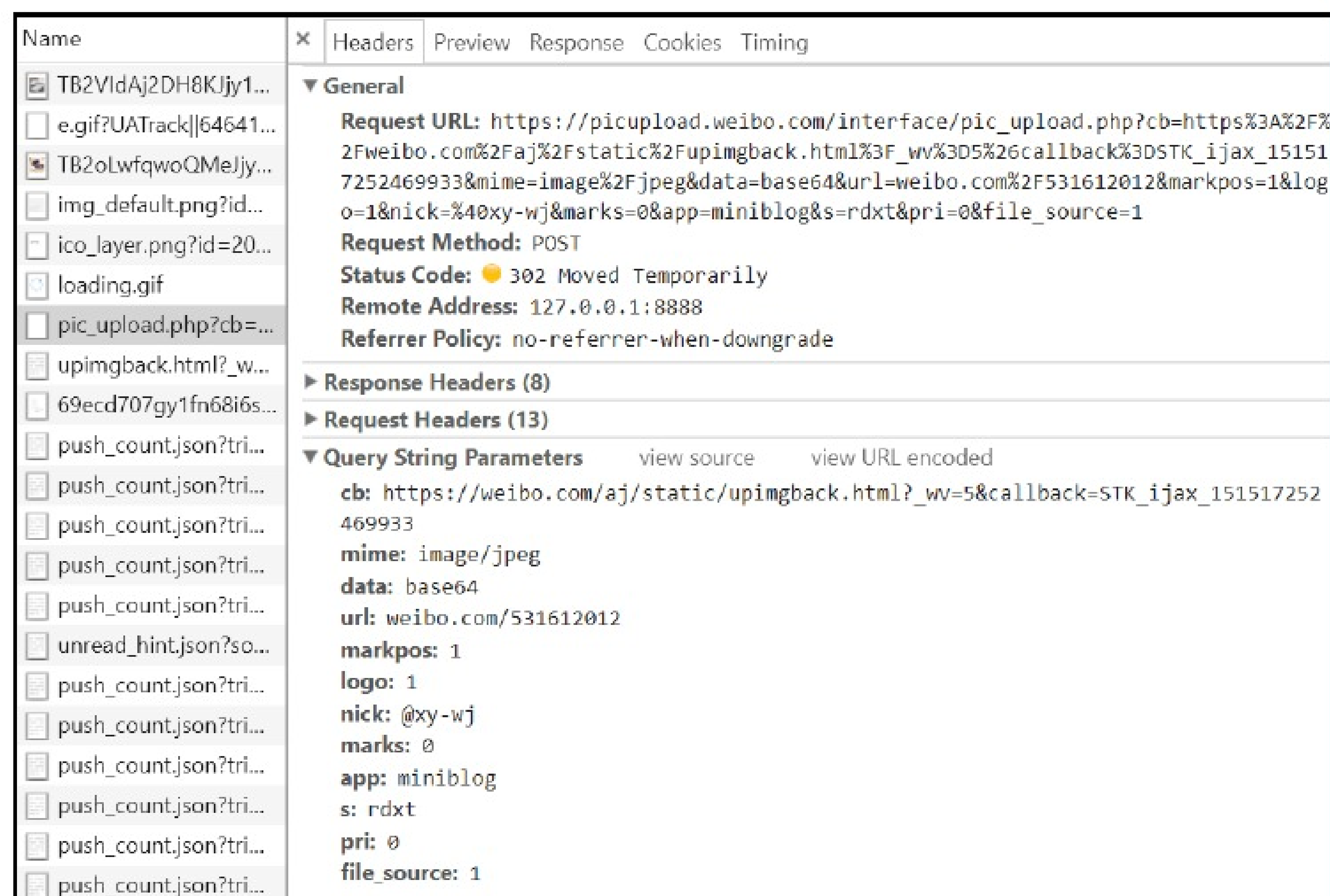


图 20-20　图片添加信息

从图 20-20 的请求信息分析，请求链接是 GET 请求，请求方法是 POST 请求，而且请求参数已在请求链接上，说明该请求 POST 的数据不是请求参数，而是 POST 图片文件，为了进一步验证猜想，我们使用 Fiddler 分析该请求信息，如图 20-21 所示。

```
POST https://picupload.weibo.com/interface/pic_upload.php?cb=https%3A%2F%2Fweibo.com%2Faj%2Fstatic%2Fupimgback.html%3F_wv%3D5%
Host: picupload.weibo.com
Connection: keep-alive
Content-Length: 368765
Cache-Control: max-age=0
Origin: https://weibo.com
Upgrade-Insecure-Requests: 1
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3218.0 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Referer: https://weibo.com/531612012/home
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9
Cookie: SINAGLOBAL=6464114471908.122.1513049031838; login_sid_t=bb7b65365f5c70699b9da61abf66bca5; cross_origin_proto=SSL; _s_t
```

b64_data=iVBORw0KGgoAAAANSUhEUgAAAu4AAAd2CAYAAABSEChDAAAACXBIWXMAAAsTAAALEwEAmpwYAAAKTWlDQ1BQaG90b3Nob3AgSUNDIHByb2ZpbGUAAHjan
2FblqWfW8tQz6yKf0d9Z5DdQxZWpsJRNB2RVAxJ1ETGa3mptYGhZUFPIx%2BWR2FSounRxVudoiihg2TZVpKtnAKAP4ToRl0dRzy30C1XLPERDlef38iaBTMzErnqa
2FDjcUyzAKfkPNFVaXC4IfjNfShY5eGDE8xdH3UQogGhloq0IMMATqdJgtrALUFqKVN0IF55B1VBaLKMzGeKYq3nWDZWMMIRPWuKLemINhRwWclD1aZ%2BBNqJBnrD
2ByZZjSaI7OOFwWqSbpc5rO5z9QxyguNuMZoC71CKacbkYw9VnJjYb9WJNFIGAXQsNqy5MNozyVAWpCV2z9%2BTO7nSBL42CGz4XFW4VXDT29u3QTtFGFDuA0KQgOk
2BaqefuZJA9wfe%2FQ18nu%2F8w%2BoOREUub0S7%2Fyu75ff810%2FkPzbVZDF%2FtZcnoQFtWe8BfTu%2Fa4ixwni%2FaDV%2FVOTNUuqp86A%2BysJuG9ZnEbjc
2F6hHIaK5J2mbLRj6zOccNjcUly7HbfHavXxIVk6jlElpfqseTPVeQnrewKnqbAAJqIVtXqWR8U6onc9FS6%2BrO%2Fj3Ihc8XORLANU21K2kwKADQAMHOHyphb27F
2BzB23Ti7xS6yxOb8NaYnbJIsuNT4RVciPZ5QJt3emYNiRtIrIuzTeNLy0ZpKVMnVCdiUDa6POKrJm%2B4M2XRL5p2wa8zg1YAn4psQ5aacvhlUkyUrh7czc5DmEMw
2BJVaP9PZdpQ1bCxPnaOpouenITpHxUFRPVcS4kjqBnuqHa2pnqGcFKONoYmBTiC7aL8jwwcZlH755d1bkQ49K5XC2Mrf2gotqv1DXFVJ%2Fn7ly26SCxI49pyNr2N
2FQGUKwqOIra9PrJtPrJyvsjVgHCV78wZYdBmIA6dWLxIbFo1Erdt9doi8OCOYph1Ecmiio Wyyody2xOk10gk3kuru9QDR3L6ORJVXuQEnxrsG%2BcPoYRXjIs6s%2
2BfPXX389rVy5MhHjt9xyC1166aUzfi0A4OSBRURwGAAAAAAAAFjYIMcdAAAAAAAACHcAAAAAAAAAhDsAAAAAAAAQ7gAAAAAAAAAIdwAAAAAAAAACEOwAAAAAAAABDuA
2B9dZbaT87ceKERkdHGSQAeIBQ4w4sk%2B3bt2vPnj2yWqOpl2%2FatEn%2F%2Fb%2F%2Fd%2F3N3%2FxNWigrJMDdvXtXg4ODcrlc6uzslMViKflruHz5sgKBgJ58
2F8Iq%2Fr79%2B%2Fv6i69%2FlKtRGSIZFIaHh4WIFAQO3t7XI4HIrFYmboTp793rJli4aHh%2FXJJ5%2Boo6Mj430Zdedut1sul0sbN240x2o%2Bm82m3bt3mzPgd
2BNyVql97W1tb1rKDQmQKfUbZSqnLhH73u9%2BZvyvzu4I4nU5VV1frnXfeSTvbcfHiRVmtVm3ZsiXr75IRGo2f19TUaOfOnXI4HGpra1MgENDVq1cVCATM64yOjub
2FqWss%2FPCHP6z4PgHA7XbLz6Ge3Lp1q%2BgzVrMhhFC4E0LIAkcb75xKpRAKhVTCvZRoHh0d1RXu9UywPHfuHK5evVrzdW7evFn2mHrXkc%2FlchVfr5HhM6lUqi
2B594Pb7S65P4d2rNQasgvFIUAIhTt5JNi4c5OampoavuyuHKirrSozV1gsFmzfvh0XLlwwFeddKvRBL6zAbPJZLXR3d%2BPSpUvo6Oioeina4XDIIUaFQmHeRftc9
2BfkSjUbz%2F%2FVv43d%2F93aq8snOFGS9epWzfvh1XrlzR%2FdvixYuxceNGlSi4ceMGAGDZsmWmv0OSJOTzecOVgWg0inQ6rRKN4XAYoVCoZjEYi8XqOk7btm1D
2Frs1q1bRd61QCCgOkYvVMescFHGL9dSOanaGPdcLgebzYZCoaDy7Jmtga2MCRdtqOdhFHtEVEq92qdSg12Z3yDya5Qx79V6%2Fi0WiyosRYvdblcZmOPj46q%2FL1

图 20-21　图片添加信息

从图 20-21 看到，图片上传 POST 的数据是 b64_data，该数据是使用 base64 对图片的字节流加密而成的。综合上述分析，微博发布需要实现两部分功能：第一是实现图片上传，获取图片 id；第二是根据条件判断选择微博发布方式。代码如下：

```python
import base64
import time
#获取上传图片 id
def upload_pic(session, watermark, nick, file_list=[]):
    pic_id_list = []
    #判断图片数量是否在 1～9 之间
    if len(file_list)>0 and len(file_list)<10:
        for i in file_list:
            url = 'https://picupload.weibo.com/interface/pic_upload.php?cb=
                  https%3A%2F %2Fweibo.com%2Faj%2Fstatic%2Fupimgback.
                  html%3F_wv%3D5%26callback%3DSTK_ijax_'+ str(int(time.
                  time()*100000))+'&mime=image%2Fjpeg&data=base64&url=weibo.
                  com%2F'+ watermark+'&markpos=1&logo=1&nick=%40'+
                  nick+'&marks=0&app=miniblog'
            #图片以字节数据流读取，然后以 base64 加密
            files={'b64_data':base64.b64encode(open(i, "rb").read())}
            #上传文件
            r = session.post(url, files=files)
            print(r.text)
            # 获取图片 id
            get_picid = eval(r.text.split('</script>')[1])['data']
                           ['pics']['pic_1']['pid']
            pic_id_list.append(get_picid)
    return pic_id_list

#发送微博，pic_id_list 是上传图片 Id 列表
def send(session,watermark, location, value, addtime='',pic_id_list=[]):
```

```
    # 构建请求头
    headers = {'Referer':'http://weibo.com/'+str(watermark)+'/home',
               'user-agent':'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
                             Gecko/20100101 Firefox/41.0'}
    # 构建请求参数
    data = {'location': location, 'text': value, 'appkey':
            '', 'style type': '1', 'pic id': '','tid': '',
            'pdetail': '','gif ids':'','addtime':addtime,
            'rank': "0", 'rankid': '', 'module': 'stissue',
            'pub type': 'dialog', 'pub source': 'main ', ' t': '0'}
    # 发送图片
    if pic id list:
        pic id=''
        for i in pic id list:
            pic id += i+'|'
        # 去除最后的"|"
        if pic_id[-1]=='|':
            pic id=pic id[0:len(pic id)-1]
        data['updata img num'] = str(len(pic id list))
        data['pic id'] = pic id
    # 构建 URL
    url='https://www.weibo.com/aj/mblog/add?
          ajwvr=6&  rnd=%s' %(int(time.time()*1000))
    r = session.post(url, data=data, headers=headers)
    if r.status code==200:
        return True
    else:
        return False
```

上述是本节实现的功能代码，存放在文件 weibo_send.py 中，整段代码由以下两个函数组成。

- upload_pic()：实现图片上传。函数参数 session、watermark、nick 和 file_list:
  - session 是带有用户登录状态的会话对象。
  - watermark 和 nick 是用户信息。
  - file_list 是图片列表，列表元素是图片路径。
- send(): 实现微博发布。函数参数有 session、watermark、location、value、addtime 和 pic_id_list:
  - session 是带有用户登录状态的会话对象。
  - watermark 和 location 是用户信息。
  - value 和 addtime 是发布内容和发布时间。
  - pic_id_list 是函数 upload_pic()返回的图片 id 列表。

send()功能说明如下：

（1）需要重新设置请求头并加入 Referer 信息，否则会导致发送失败，因为网站做了检测 Referer 的反爬虫机制。

（2）请求参数合并了三种不同的发布方式，例如只发布文字内容，只需将参数 pic_id 和 addtime 的值设置为空即可。若发布图片，在设置 pic_id 的参数值时，则会相应地创建参数 updata_img_num。

（3）请求链接最后的一串数字是当前时间的时间戳再乘以 1000 后取整所得。

代码与 16.4 节的运行方式一样，打开修改微博登录文件 weibo_login.py，代码如下：

```
if   name   == "  main  ":
    # 构造请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
             Gecko/ 20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent
    }
    # 代理 IP
    proxies = {}
    # 新建会话
    session = requests.session()
    # 第三方平台账号、密码
    yundama username = 'xxxx'
    yundama_password = 'xxxx'
    user info = login('13435423143','xxxx')
    # 导入微博发布模块
    from weibo send import upload pic,send
    # 获取用户信息
    watermark = user info['watermark']
    nick = user info['nick']
    location = user info['location']
    # 设置图片列表
    file list=['aa.png','bb.png']
    # 获取图片 id 列表
    pic id list = upload pic(session,watermark,nick,file list)
    # 发布微博
    send(session, watermark, location, "Python 爬虫", addtime='',
          pic_id_list=pic_id_list)
```

20.6 关 注 用 户

在微博中关注用户有两种方式：

（1）在用户的某条微博上，在将鼠标移到用户头像时所弹出的窗口中单击“关注”。

（2）在微博用户的首页单击“关注”。

两种关注方式的请求链接是一样的，区别在于请求参数的差异。本项目主要实现第二种关注方式，在浏览器上捕捉其请求信息，如图 20-22 和图 20-23 所示。

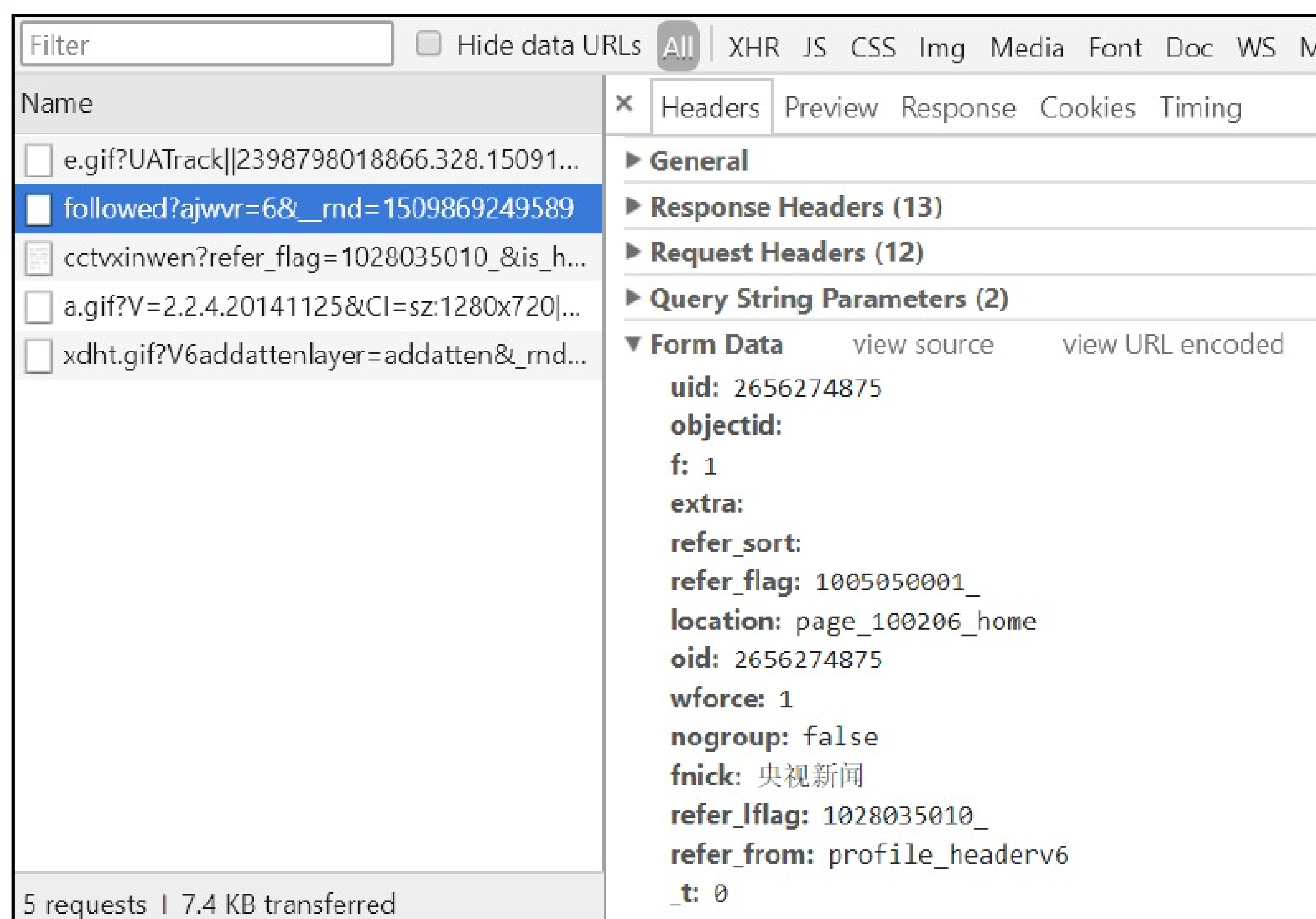


图 20-22　关注用户时的请求信息

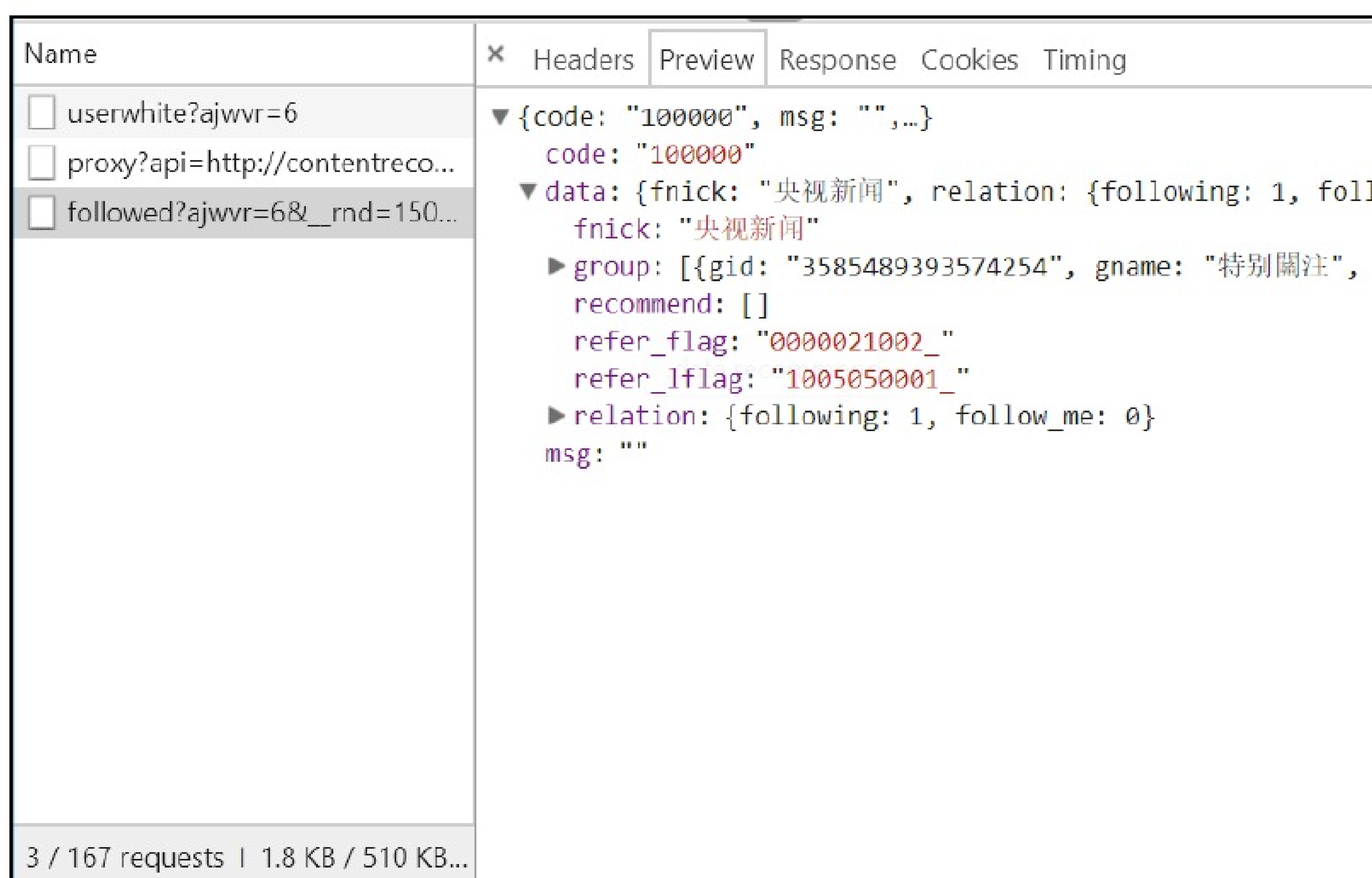


图 20-23　关注用户时的响应内容

从图 20-22 的请求参数分析可得，参数 uid、location、oid 和 fnick 是被关注用户的信息，暂时无法得知被关注用户信息的来源。其余的参数是固定不变的。

从图 20-23 的响应内容分析可得，用户被关注成功之后，网站主要返回 JSON 数据。观察数据内容，可从“code”的值来判断是否关注成功。

进一步核实被关注用户信息的来源，以“央视新闻”的微博为例，分析查找浏览器在“央视新闻”的微博首页所捕捉的请求信息，最终在 Doc 标签找到该微博信息，如图 20-24 所示。

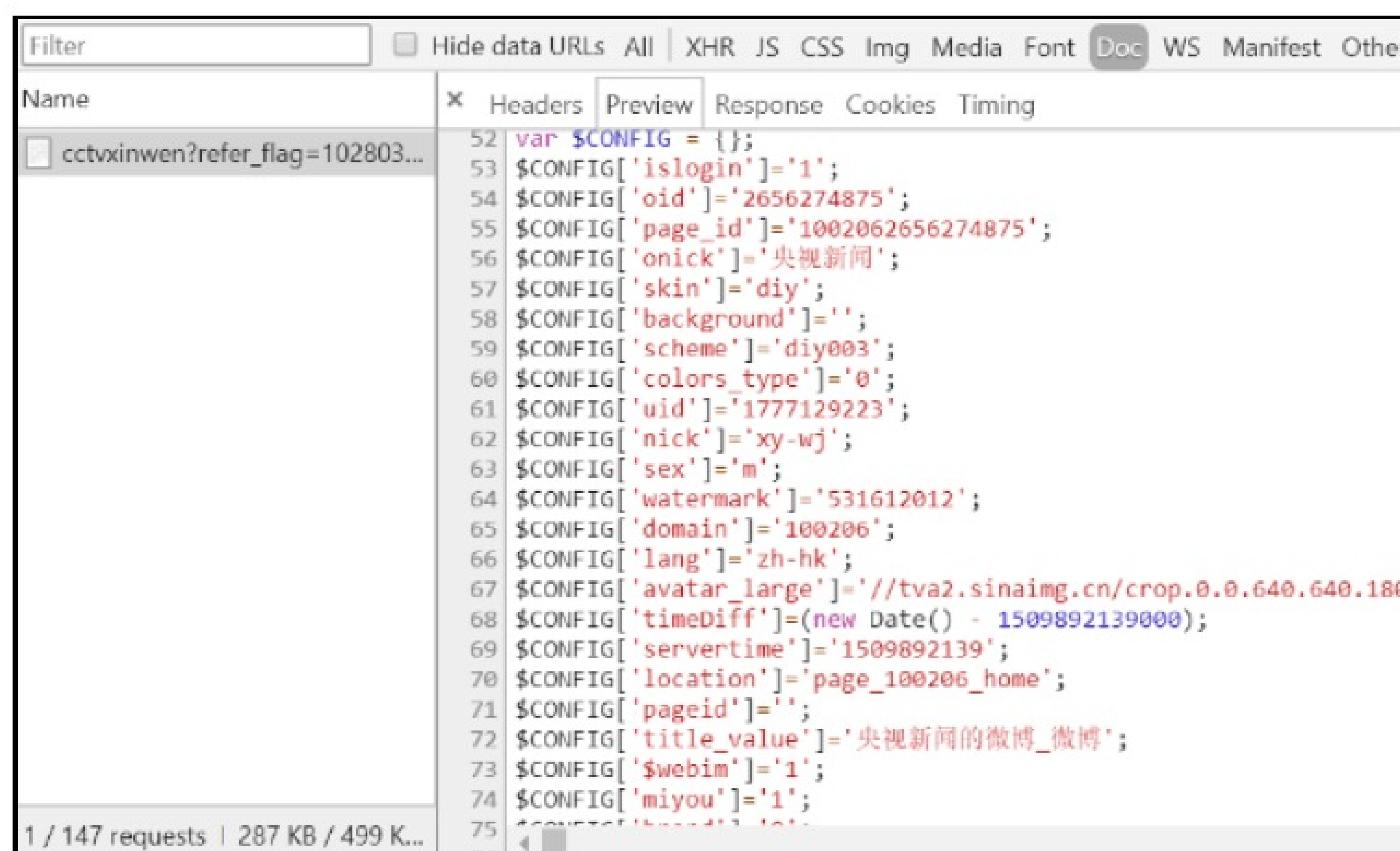


图 20-24　被关注用户的首页信息

从图 20-22 和图 20-24 的数据对比得出，图 20-22 的参数 uid、location、oid 和 fnick 分别对应图 20-24 的 oid、location、oid 和 onick。

综合上述分析，实现代码如下：

```
import time
# 关注微博，session 是用户登录后的会话，follow_url 是关注用户的微博主页
def follow_weibo(session, follow_url):
    # 构建请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
             Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent,
        'Referer': follow_url
    }
    follow_info = {'oid': '', 'onick': '', 'location': ''}
    r = session.get(follow_url)
    response = r.text
    follow_info['oid'] = response.split("$CONFIG['oid']= '")
                         [1].split("';")[0]
    follow_info['onick'] = response.split("$CONFIG['onick']= '")
                         [1].split("';")[0]
    follow_info['location'] = response.split("$CONFIG['location']= '")
                         [1].split("';")[0]
    # 关注 URL，参数__rnd 为时间戳
    url = 'https://www.weibo.com/aj/f/followed?
           ajwvr=6&__rnd=' + str(int(time.time() * 1000))
    data = {
        'uid': follow_info['oid'],
        'objectid': '',
        'f': '1',
        'extra': '',
        'refer_sort': '',
        'refer_flag': '1005050001 ',
        'location': follow_info['location'],
        'oid': follow_info['oid'],
        'wforce': '1',
```

```
        'nogroup': 'false',
        'fnick': follow info['onick'],
        'refer lflag': '',
        'refer_from': 'profile_headerv6',
        ' t': '0'
    }
    r = session.post(url, data=data, headers=headers)
    # 判断是否关注成功
    if (r.json()['code']) == '100000':
        return (follow_info['onick'] + '关注成功')
    else:
        return (follow_info['onick'] + '关注失败')
```

上述是本节实现的功能代码，存放在文件 weibo_follow.py 中，代码说明如下：

（1）函数 follow_weibo 的参数分别是带有用户信息的会话对象和被关注的微博首页链接。
（2）重新构建请求头，主要在发送关注用户的请求时所使用。
（3）访问被关注用户的首页，获取被关注用户的信息。
（4）对获取的数据构建请求参数，实现发送用户关注请求。

代码与 20.5 节的运行方式一样，打开修改微博登录文件 weibo_login.py，代码如下：

```
if   name   == "  main  ":
    # 构造请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
             Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent
    }
    # 代理 IP
    proxies = {}
    # 新建会话
    session = requests.session()
    # 第三方平台账号、密码
    yundama username = 'xxxx'
    yundama password = 'xxxx'
    user_info = login('13435423143','xxxx')
    # 导入关注用户模块
    from weibo follow import follow weibo
    # 关注用户的首页链接
    follow url = 'https://weibo.com/renminwang'
    status = follow weibo(session, follow url)
    print(status)
```

20.7　点赞和转发评论

本节主要实现两个功能：点赞和转发评论。两者实现方式和 20.6 节的实现方式大致相同，主要在对方的微博首页实现。

在浏览器上访问某微博的用户首页，以“有妖气原创漫画梦工厂”为例（https://weibo.com/u17t），

在该微博用户的第一条微博中单击“点赞”按钮，开发者工具所捕捉的请求信息如图 20-25 所示。

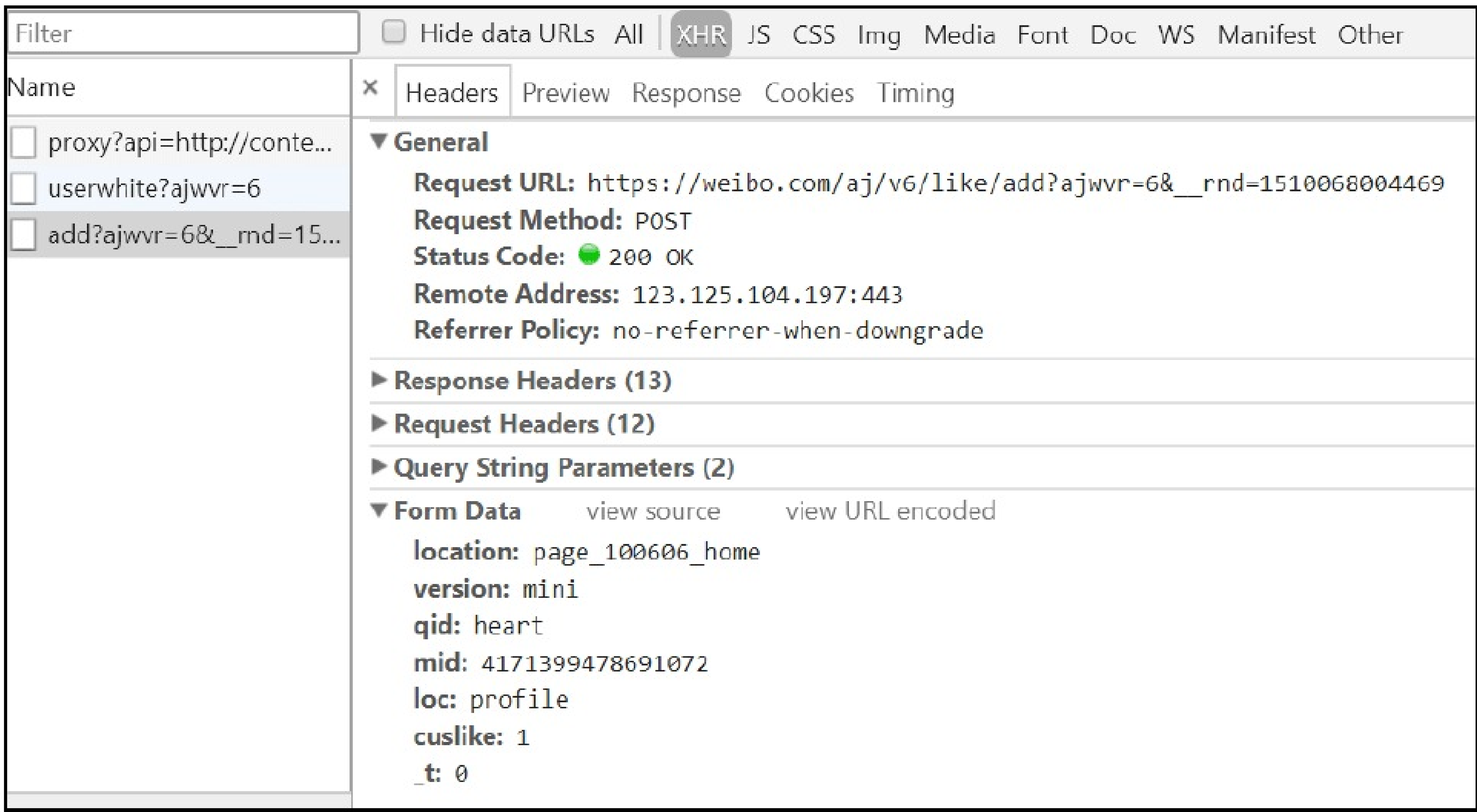


图 20-25 点赞微博请求信息

从图 20-26 的请求分析可知，请求链接的_rnd 是当前时间的时间戳乘以 1000 再取整所得，请求参数分析如下：

（1）参数 location 代表被点赞的微博用户信息，数据来源可在 Doc 标签返回的 HTML 内容中找到。

（2）参数 mid 数据来源无法得知，点赞不同的微博，其参数值随之变化。

（3）其余参数值固定不变。

根据参数 mid 的变化规律得知，不同微博的数据会随之变化，那么参数 mid 可能用于标识微博的唯一性。为了验证猜想是否正确，我们分析浏览器所捕捉到的请求信息，最终在 Doc 下找到 mid 的参数值，如图 20-26 所示。

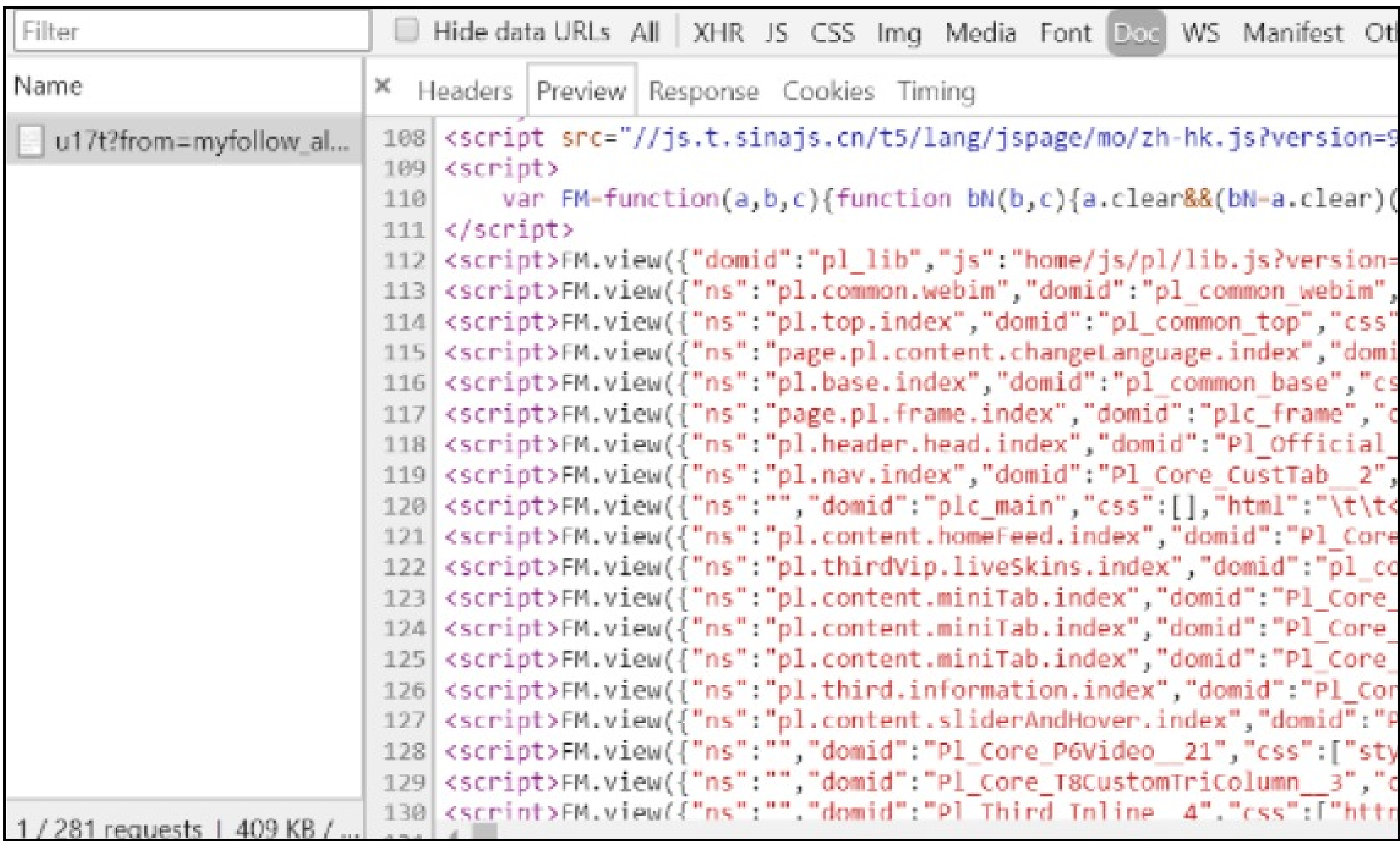


图 20-26 查找请求参数

在 Doc 标签返回的 HTML 内容中快速查找（Ctrl+F）参数值（4171399478691072），在 HTML 里的 JavaScript 代码中找到参数 mid，而且参数 mid 是重复出现的。因此可以确定，参数 mid 可在网站返回的 HTML 中找到。综合上述分析，实现代码如下：

```
import re
import time
# session 是会话对象，like url 是用户首页
def like_weibo(session, like_url):
    # 获取点赞用户的前 16 条微博
    r = session.get(like url)
    # 获取 location
    location = r.text.split("$CONFIG['location']='")[1].split("';")[0]
    # 获取 mid
    mid list = re.findall(r'mid=(.\d+)&name', r.text, re.S)
    # 构建请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
            Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent,
        'Referer': like url
    }
    # 点赞功能，默认点赞第一条微博
    url = 'https://weibo.com/aj/v6/like/add?ajwvr=6&  rnd=' +
          (str(int(time.time() * 1000)))
    data = {
        'location': location,
        'version': 'mini',
        'qid': 'heart',
        'mid': mid list[0],
        'loc': 'profile',
        'cuslike': '1',
        ' t': '0'
    }
    r = session.post(url, data=data, headers=headers)
    # 根据返回内容判断是否成功
    if (r.json()['code']) == '100000':
        return ('点赞成功')
    else:
        return ('点赞失败')
```

点赞功能定义为函数 like_weibo()：函数参数 session 是会话对象；like_url 是被点赞用户的首页链接。函数实现的功能如下：

（1）访问被点赞用户的首页链接，获取用户的 location 信息和 mid_list。mid_list 是当前用户的前 16 条微博 mid 组成的列表。

（2）构建请求头，作为发送点赞请求的请求头。如果不加请求头，该请求就会被服务器视为非法请求，因为服务器会对请求头的 Referer 进行检查，这是一种反爬虫机制。

（3）发送点赞请求，将获取的 location 和 mid_list 作为请求参数，mid_list 默认取第一位元素，即默认点赞第一条微博。

（4）针对请求后的响应内容判断是否点赞成功。

完成微博点赞功能后，接着完成转发评论功能，该功能的实现方式和点赞功能类似。以上述微博用户首页为例，单击转发该用户的第一条微博，勾选“同时评论”选项，在开发者工具看到该请求信息，如图 20-27 所示。

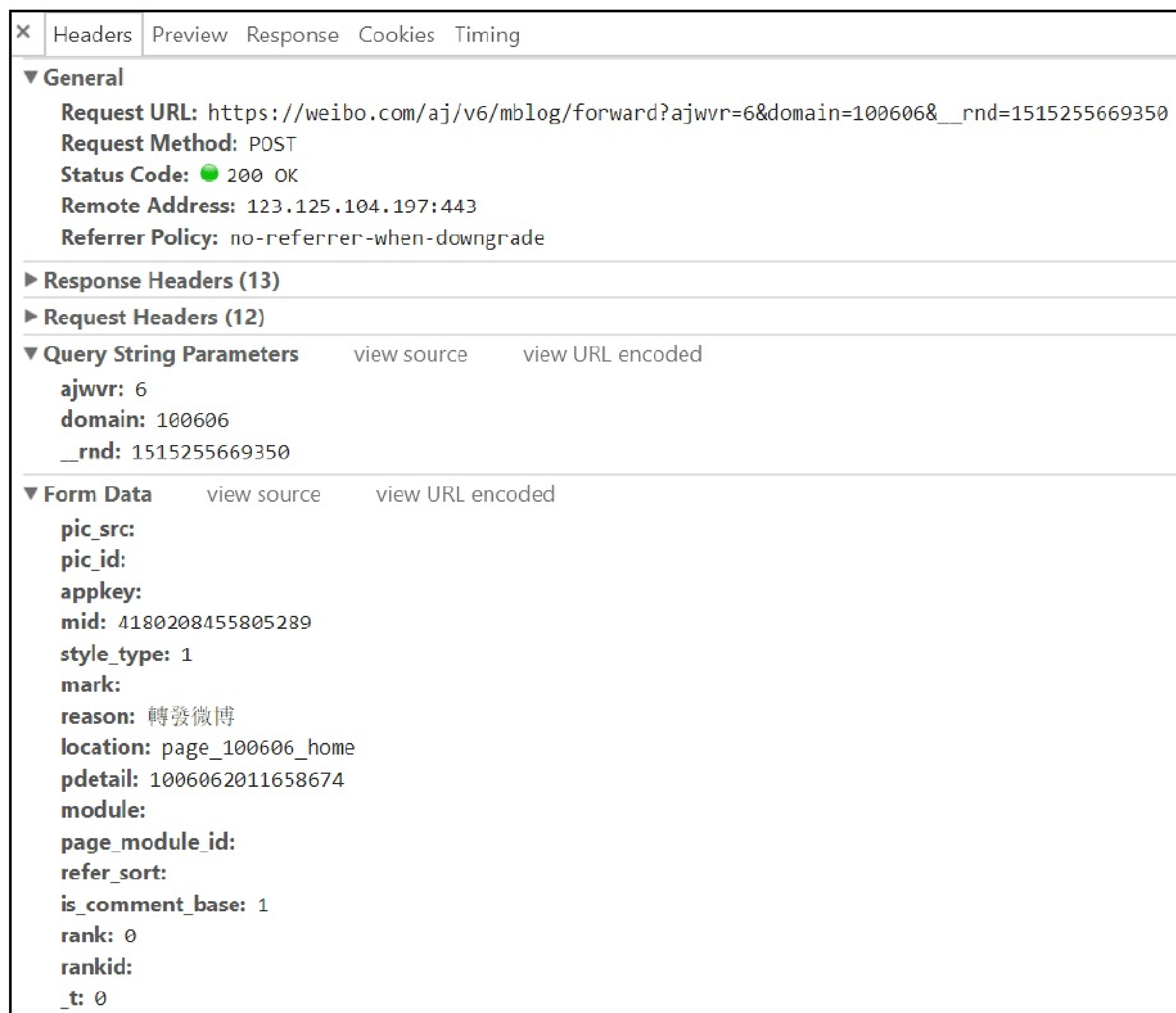


图 20-27　转发评论的请求信息

从请求信息分析可得，请求链接的_rnd 是当前时间的时间戳乘以 1000 再取整所得，domain 是被转发的用户信息，请求参数分析如下：

（1）参数 location 和 mid 与点赞功能的请求参数一致。

（2）参数 reason 是转发内容。

（3）参数 pdetail 无法确定。

（4）参数 pic_id 与 20.5 节的请求参数 pic_id 一致。

（5）参数 is_comment_base 代表转发时的“同时评论”选项。

（6）其余参数值固定不变。

为了确定参数 pdetail 的数据来源，查找分析浏览器所捕捉到的请求信息，最后在 Doc 下找到该参数的数据来源，该参数代表被转发微博的用户信息，如图 20-28 所示。

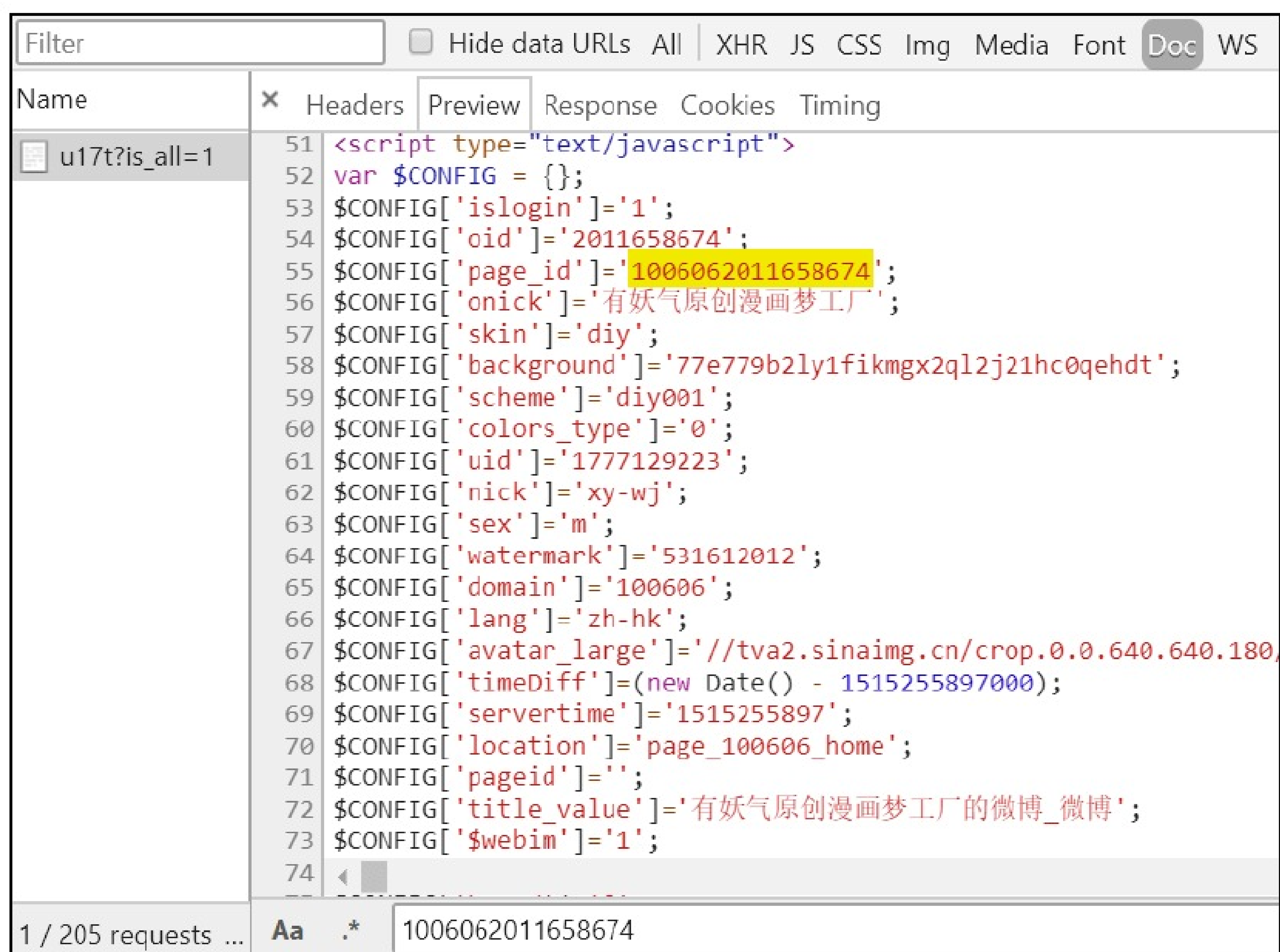


图 20-28　请求参数信息

综合上述分析，实现代码如下：

```
# 转发评论微博
def forward weibo(session, forward url, reason):
    # 获取点赞用户的前 16 条微博
    r = session.get(forward url)
    # 获取 location
    location = r.text.split("$CONFIG['location']='")[1].split("';")[0]
    page_id = r.text.split("$CONFIG['page_id']='")[1].split("';")[0]
    domain = r.text.split("$CONFIG['domain']='")[1].split("';")[0]
    # 获取 mid
    mid list = re.findall(r'mid=(.\d+)&name', r.text, re.S)
    # 构建请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
              Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent,
        'Referer': forward url
    }
    # 转发评论
    url = 'https://weibo.com/aj/v6/mblog/forward?ajwvr= 6&domain='+ domain
          +'&  rnd=' + (str(int(time.time() * 1000)))
    data = {
        'pic src': '',
        'pic id': '',
        'appkey': '',
        'mid': mid list[0],
        'style type': '1',
        'mark': '',
        'reason': reason,
        'location': location,
```

```
        'pdetail': page id,
        'module': '',
        'page module id': '',
        'refer_sort': '',
        'is comment base': '1',
        'rank': '0',
        'rankid': '',
        ' t': '0'
    }
    r = session.post(url, data=data, headers=headers)
    # 根据返回内容判断是否成功
    if (r.json()['code']) == '100000':
        return ('转发成功')
    else:
        return ('转发失败')
```

转发评论功能定义函数 forward_weibo()：函数参数 session 是会话对象；forward_url 是被转发评论用户的首页链接；reason 是转发的评论内容。函数的功能逻辑与点赞功能大致相同，此处不做详细讲解。

将上述函数 like_weibo()和 forward_weibo()保存在文件 weibo_forward.py 中，代码与 20.6 节的运行方式一样，打开修改微博登录文件 weibo_login.py，代码如下：

```
if  name   == "  main  ":
    # 构造请求头
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
               Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent
    }
    # 代理 IP
    proxies = {}
    # 新建会话
    session = requests.session()
    # 第三方平台账号、密码
    yundama username = 'xxxx'
    yundama password = 'xxxx'
    user info = login('13435423143', 'xxxx')
    # 导入点赞和转发评论模块
    from weibo forward import forward weibo, like weibo
    url = 'https://weibo.com/u17t'
    # 点赞
    result = like weibo(session, url)
    print(result)
    # 转发评论
    result = forward_weibo(session, url, 'Python 网络爬虫')
    print(result)
```

20.8 本 章 小 结

通过本章的学习，读者要着重掌握以下知识点：

1. 项目实现的功能

- weibo_login.py：微博用户登录，同时也是程序运行文件。
- weibo_verify_code.py：第三方平台 API，实现验证码识别。
- weibo_collect.py：根据关键字搜索并采集热门微博。
- weibo_send.py：发布微博。
- weibo_follow.py：关注用户。
- weibo_forward.py：微博点赞和转发评论。
- data.csv：存储采集数据。
- 文件夹 video 和 image：分别存储采集的视频和图片。

2. 微博登录实现难点

（1）账号密码的加密处理。加密方法一般在 JS 代码中能直接找到，开发人员需要对 JS 代码解读分析。

（2）带验证码登录和普通登录的区别，程序运行要根据当前的登录模式而做出响应的登录处理。

（3）用户登录成功后获取用户信息。

（4）第三方平台识别验证码。

3. 关键字搜索热门微博实现难点

（1）关键词 URL 编码处理，由 urllib.parse.quote()实现。

（2）视频文件的 URL 地址分析以及提取。

（3）多线程下载图片和视频。

4. 发布微博实现难点

（1）图片上传分析以及功能实现。

（2）分析三种微博发布方式的异同。

5. 关注用户、点赞和转发评论实现难点

（1）分析请求参数含义以及来源。

（2）构建请求头。

第 21 章

实战：微博爬虫软件开发

21.1 GUI 库及 PyQt5 的安装与配置

在本项目中，主要讲解如何使用 PyQt5 实现软件开发，首先介绍 GUI 库和 PyQt5 安装与配置。

21.1.1 GUI 库

Python 提供了多个图形开发界面的库（GUI 库），常用的 GUI 库有：

- Tkinter（也叫 Tk 接口）是 Tk 图形用户界面工具包标准的 Python 接口。Tk 是一个轻量级的跨平台图形用户界面（GUI）开发工具，可以运行在大多数 UNIX 平台、Windows 系统和 Mac 系统中。
- wxPython 是 Python 语言的一套优秀的 GUI 图形库，允许 Python 程序员很方便地创建完整的、功能健全的 GUI 用户界面。wxPython 作为优秀的跨平台 GUI 库，以 wxWidgets 的 Python 封装和 Python 模块的方式提供给用户。
- PyQt 是 Qt 库的 Python 版本。PyQt3 支持 Qt1 到 Qt3，PyQt4 支持 Qt4，PyQt5 支持 Qt5。PyQt 的首次发布是在 1998 年，当时叫作 PyKDE，因为那时 SIP 和 PyQt 没有分开。PyQt 是用 SIP 写的，提供 GPL 版和商业版。
- Kivy 是一个开源工具包，是能够使用相同源代码创建的程序，并且可以跨平台运行。它主要关注创新型用户界面开发，如多点触摸应用程序。Kivy 还提供一个多点触摸鼠标模拟器。Kivy 当前支持的平台包括 Linux、Windows、Mac 和 Android，拥有能够处理动画、缓存、手势和绘图等功能。Kivy 还内置许多用户界面控件，如按钮、摄影机、表格、Slider 和树形控件等。
- Flexx 是一个纯 Python 工具包，用来创建图形化界面应用程序，使用 Web 技术进行界面的渲染。可以用 Flexx 来创建桌面应用，同时也可以导出一个应用到独立的 HTML 文档。因为使用纯 Python 开发，所以 Flexx 可跨平台使用。只需要有 Python 和浏览器，Flexx 就可以运行。

21.1.2　PyQt5 安装及环境搭建

PyQt5 是一套绑定 Qt5 的应用程序框架，由 Python 语言实现，已经有超过 620 个类和 6000 个函数与方法。PyQt5 是一个运行在所有主流操作系统上的多平台组件，包括 UNIX、Windows 和 Mac OS。PyQt5 是双重许可的，开发者可以选择 GPL 和商业许可。

PyQt5 可以使用 pip 安装：

```
pip install PyQt5
```

完成 PyQt5 的安装后，接着安装图形界面的开发工具，这是能快速开发图形界面的辅助工具。如果对 PyQt5 比较熟悉，可以使用 Python 纯代码开发图形界面。

开发工具有 Qt Creator 与 Qt Designer，两者都能实现图形界面的开发。其中，后者是前者一部分功能的“阉割”。Qt Creator 包括项目生成向导、高级的 C++代码编辑器、浏览文件及类的工具，集成了 Qt Designer、Qt Assistant、Qt Linguist、图形化的 GDB 调试前端和 qmake 构建工具等。

安装 Qt Creator 可以到官方网站下载.exe 安装包（https://www1.qt.io/download/），下载安装包前需要注册和填写个人信息才行。

Qt Designer 仅支持在 Windows 安装，并且可以使用 pip 安装：

```
pip install PyQt5-tools
```

本书以 Qt Designer 作为图形界面开发工具，安装 Qt Designer 后，可以在 Python 安装目录\Lib\site-packages\pyqt5-tools 找到 designer.exe，双击并打开 Qt Designer，如图 21-1 所示。

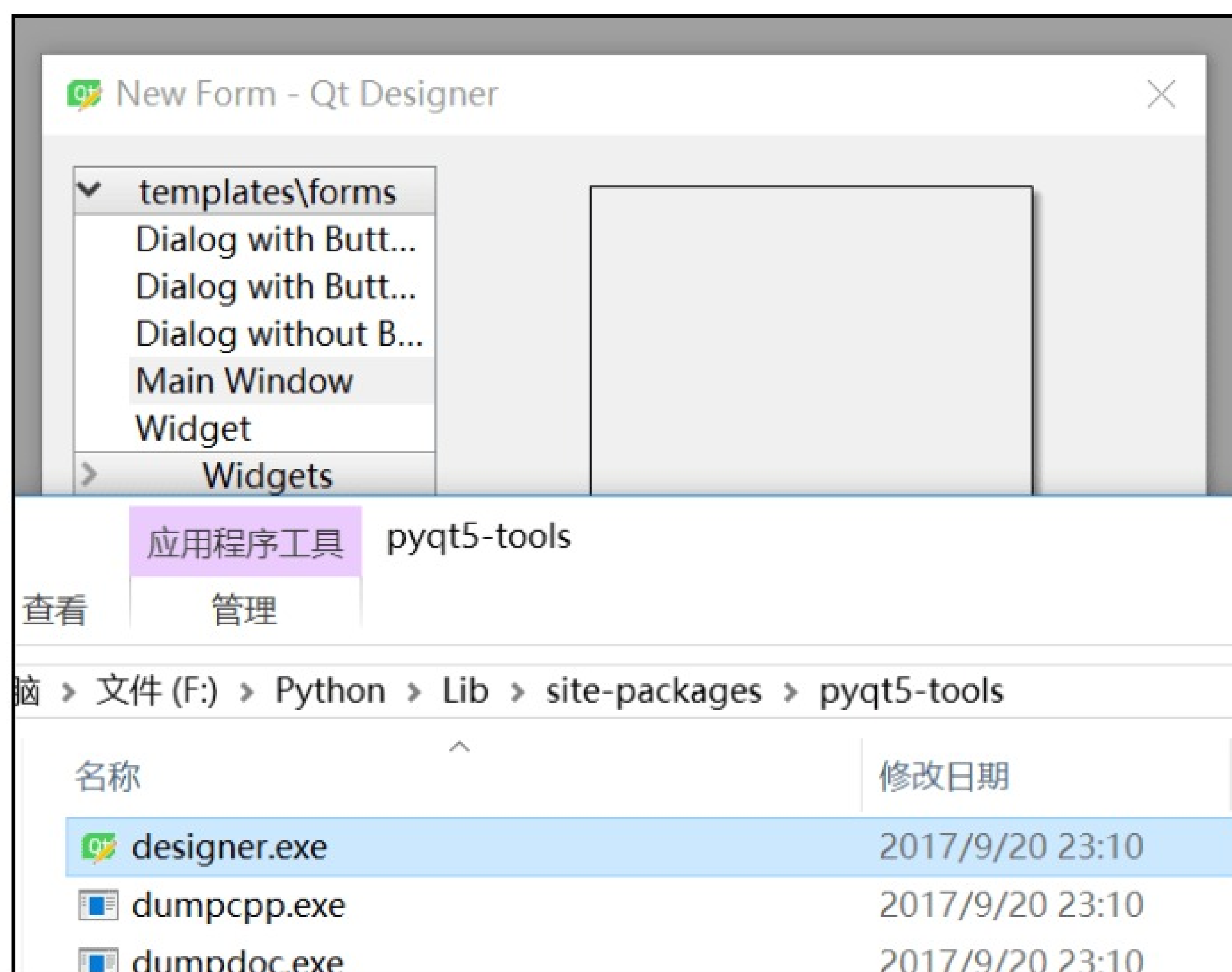


图 21-1　Qt Designer

安装 PyQt5 和 Qt Designer（Qt Creator）之后，接下来在 PyCharm 中搭建开发环境。为什么要在 PyChram 中搭建开发环境？这里由于我们使用 Qt Designer（Qt Creator）创建并生成图形界面文件，文件以 ui 为后缀名，在 Python 中无法识别该文件内容，搭建环境的目的是将 ui 文件转换成 py 文件。

不同的 PyChram 版本配置步骤有所区别，以 Windows 的 PyCharm 为例，PyChram 版本如图 21-2 所示。



图 21-2　PyChram 版本信息

配置步骤如下：

步骤01 单击“File”里面的“Settings”，找到“Tools”里面的“External Tools”，如图 21-3 所示。

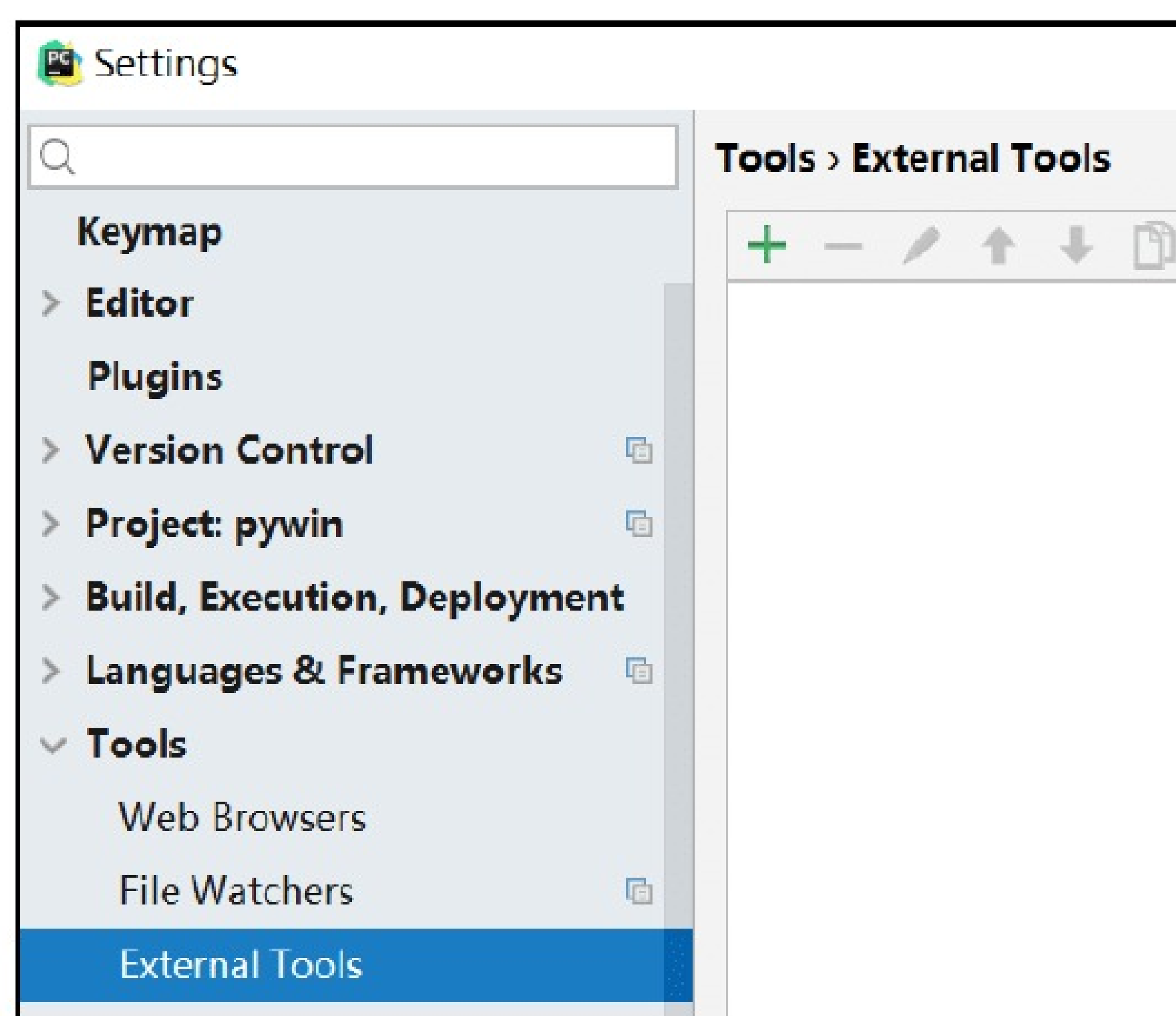


图 21-3　External Tools

步骤02 单击“Tools→External Tools”下方的“+”，新建一个 Tool，输入信息，如图 21-4 所示。

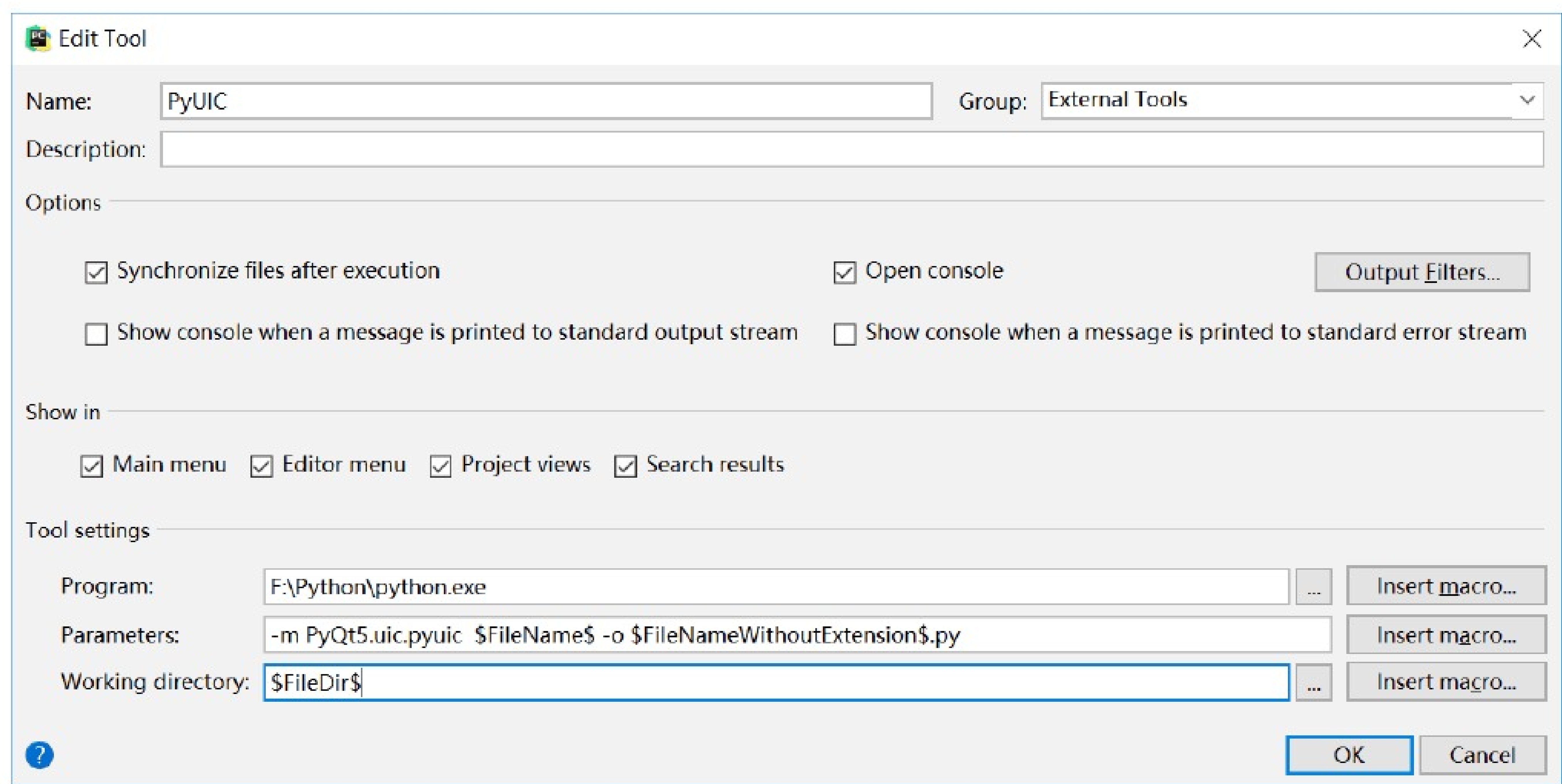


图 21-4　配置 Tools

从图 21-4 中看到，Program 的内容是 Python 安装目录的 python.exe，这是 Python 解释器；Parameters 是将 ui 文件转换为 py 文件的命令行；Working directory 是转换后生成文件的保存路径。

配置 PyChram 主要是将 ui 文件快速转换成 py 文件，不是一定要配置 PyChram 才能转换文件，也可以在 CMD（终端）界面运行 Parameters 中的命令行来实现转换。

完成了 PyQt5 和 Qt Designer（Qt Creator）的安装，并在 PyChram 中配置了文件转换工具，接下来开始讲解软件开发。

21.2　项 目 分 析

本项目是将热门微博爬取和微博发布的功能以软件的形式表示，整个软件一共有 4 个软件界面，每个界面的功能说明如下：

（1）软件主界面。爬虫软件的启动界面，界面共有三个按钮：发布、采集和相关服务，三个按钮分别进入不同的功能界面，如图 21-5 所示。



图 21-5　软件主界面

（2）相关服务界面。相关服务是让用户设置验证码打码识别及代理 IP 服务。由于微博登录可能出现验证码识别，因此需要设置第三方的验证码识别服务。此外，软件还支持多个微博账号批量发布微博，设置代理 IP 服务防止微博服务器检测并查封微博账号，如图 21-6 所示。



图 21-6　相关服务界面

（3）微博采集界面。通过关键词搜索相关的热门微博，并根据软件上的设置进行爬取。软件上设有微博账号密码、采集内容（即关键词）、采集选项以及采集页数，采集后的微博信息显示在软件右侧的表格。如图 21-7 所示。

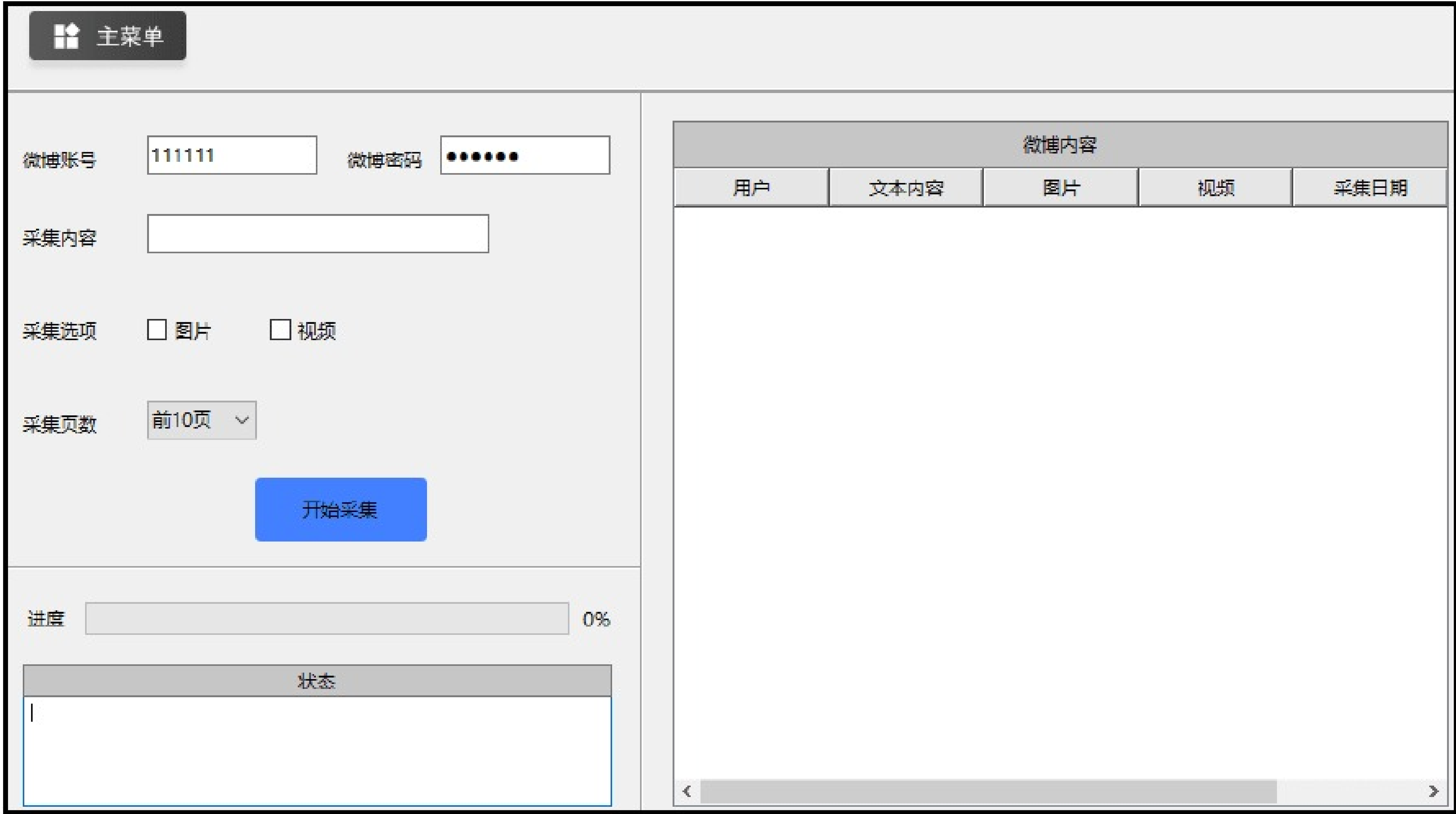


图 21-7　微博采集界面

（4）微博发布界面。支持微博内容的编辑和发布，内容编辑可在软件的右侧的表格里进行，同时支持 CSV 文件的导入和导出；软件的左侧是软件功能：服务验证、文件通道、内容编辑、发布间隔以及定时发布，如图 21-8 所示。

以上述是整个软件的 4 个界面，每个界面所实现的功能看似简单，但实现过程还是相当复杂的。不仅要熟练 PyQt5 开发，还要将爬虫的功能嵌入到软件里，通过软件来控制爬虫的爬取方式。为了让读者对项目有整体的认知，以下对项目的文件目录进行分析说明，如图 21-9 所示。

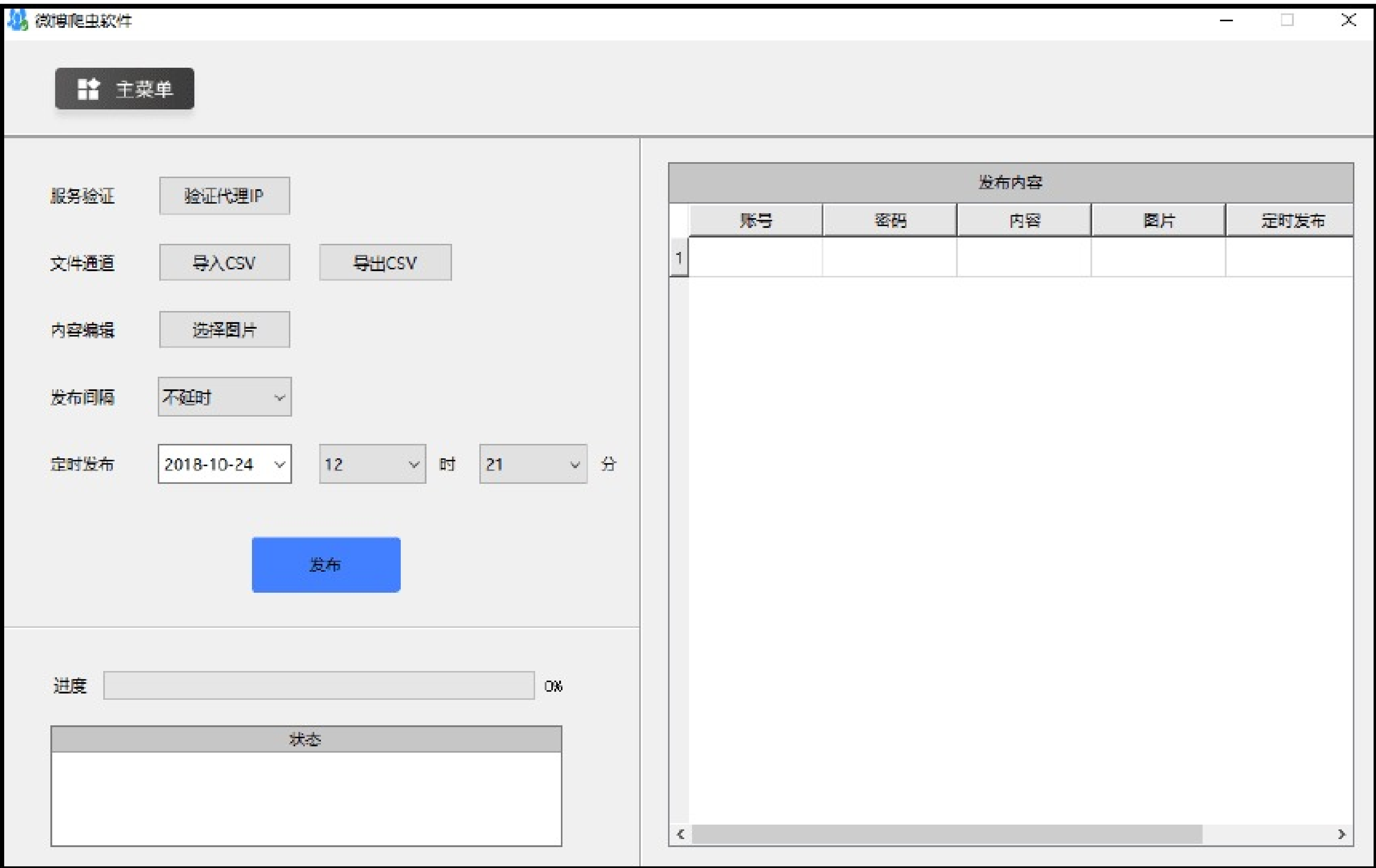


图 21-8　微博发布界面

ico	文件夹	
temp	文件夹	
collect.py	Python File	7 KB
main.py	Python File	2 KB
release.py	Python File	20 KB
service.py	Python File	5 KB
weibo.py	Python File	16 KB
weibo_collect.py	Python File	13 KB
weibo_main.py	Python File	2 KB
weibo_release.py	Python File	23 KB
weibo_service.py	Python File	7 KB
weibo_verify_code.py	Python File	5 KB
weibo_collect.ui	UI 文件	14 KB
weibo_main.ui	UI 文件	3 KB
weibo_release.ui	UI 文件	23 KB
weibo_service.ui	UI 文件	8 KB

图 21-9　项目的文件目录

整个项目的文件目录共有 16 个文件或文件夹，每个文件或文件夹的作用说明如表 21-1 所示。

表 21-1　项目文件及文件夹说明

文件或文件夹	作用
ico 文件夹	存放软件的界面图片，如背景图、按钮图标等
temp 文件夹	存放软件的 CSV 文件和配置文件
collect.py	微博采集的功能逻辑代码，如按钮的信号和槽（即触发事件）
main.py	主界面的功能逻辑代码，同时也是软件的运行文件
service.py	相关服务的功能逻辑代码，如设置代理 IP 和验证码识别的用户信息
release.py	微博发布的功能逻辑代码，设置微博发布方式，如定时发布、带图片发布等
weibo.py	定义微博登录、发布和采集的爬虫函数

（续表）

文件或文件夹	作用
weibo_collect.py	微博采集的界面设计，代码是由 weibo_collect.ui 转换而成
weibo_main.py	主界面的界面设计，代码是由 weibo_main.ui 转换而成
weibo_service.py	相关服务的界面设计，代码是由 weibo_service.ui 转换而成
weibo_release.py	微博发布的界面设计，代码是由 weibo_release.ui 转换而成
weibo_verify_code.py	第三方的验证码识别接口，用于识别微博登录的验证码
weibo_collect.ui	微博采集的界面设计，由 QT Designer 生成
weibo_main.ui	主界面的界面设计，由 QT Designer 生成
weibo_service.ui	相关服务的界面设计，由 QT Designer 生成
weibo_release.ui	微博发布的界面设计，由 QT Designer 生成

21.3　软件主界面

从软件主界面的效果图可以看出，整个界面共有三个按钮以及微博的背景图，这三个按钮分别是发布、采集和相关服务，当分别单击这三个按钮的时候，软件就会自动切换到相应的功能界面。

根据软件效果图，打开 QT Designer 设计器来设计软件主界面。在 Qt Designer 可以看到一个新建窗口的界面，有 5 种功能模板可供选择，如图 21-10 所示。

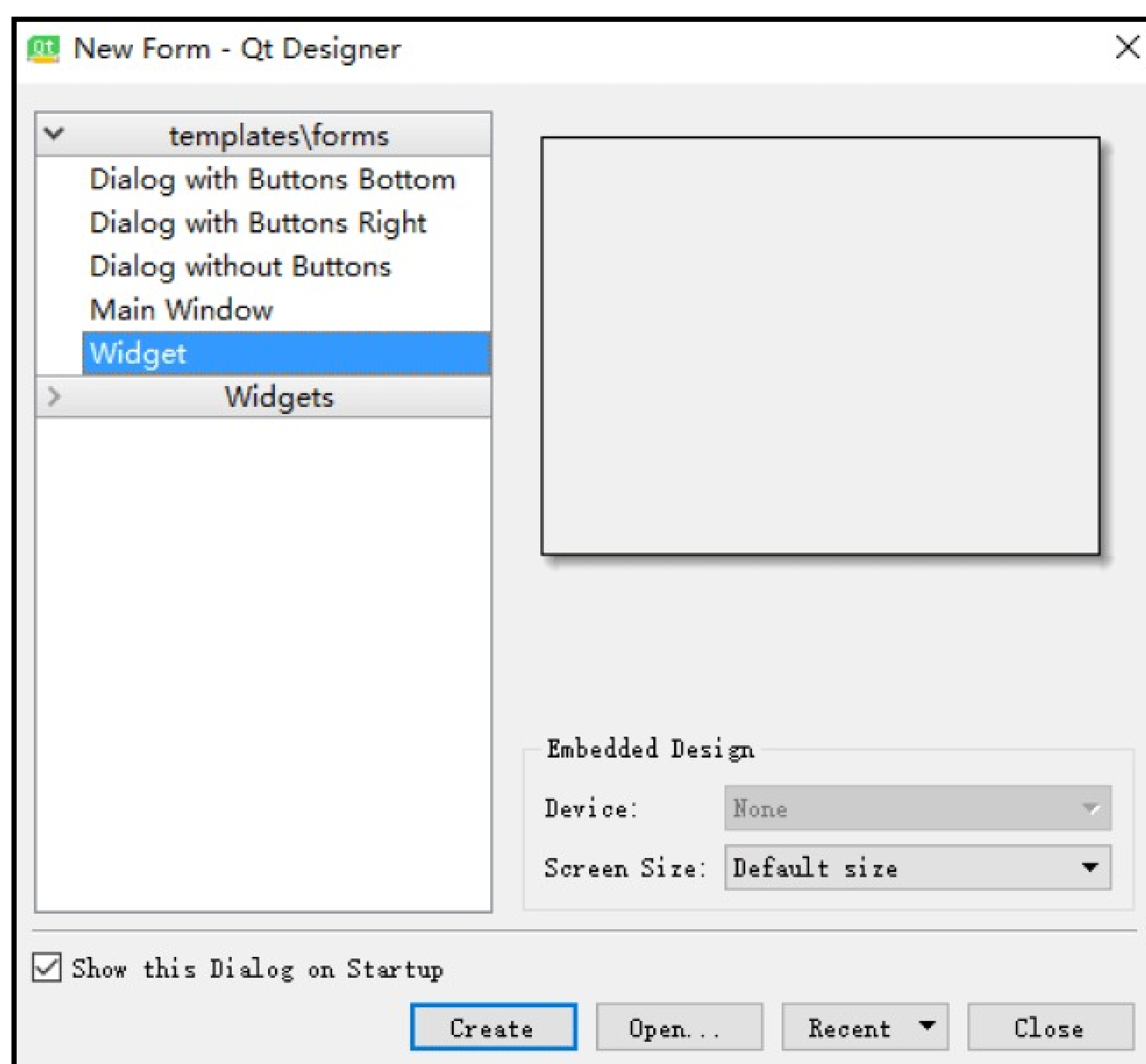


图 21-10　Qt Designer 新建窗口

虽然新建窗口提供 5 种功能模板，但实际上只有 3 种不同类型的模板，分别是 Dialog、MainWindow 和 Widget，三者作用如下：

（1）MainWindow 是主界面，一个窗口是父/子 hiearchy 的顶部，通常显示标题栏和边框。底

层窗口系统（Windows、KDE、GNOME 等）将为窗口提供策略，如标题栏/边框样式、布局和焦点等。

（2）Widget 是小部件，是屏幕上的一个矩形区域，用于显示和用户交互，包括按钮、滑块、视图、对话框和窗口等。所有窗口小部件将在屏幕上显示某些内容，许多窗口小部件也将接受来自键盘或鼠标的用户输入。“widget”一词来自 UNIX，在 Windows 中称为“控件”。

（3）Dialog 为对话框，通常是临时的，可以设置不同的标题栏外观，主要用于通知或收集输入窗口，并且底部或右侧通常具有 OK、Cancel 等按钮。

选择 Widget 并创建模版，将其作为软件的主界面，在 QT Designer 界面里分为 5 个区域，正中间区域是软件设计的界面，左右两侧是功能区域，如图 21-11 所示，功能区域的说明如下。

- 区域 1：控件区，软件的功能控件都在此区域生成，可以拖来控件到模板上实现可视化软件设计。
- 区域 2：软件的目录结构，显示模板中所有控件的类型，能帮助设计者快速找到控件。
- 区域 3：控件属性区，主要修改控件的属性。
- 区域 4：信号（Signal）/槽（Slot），信号和槽是 Qt 编程中对象间的通信机制。简单地说，就是单击按钮时候所触发的事件。单击按钮称之为信号，触发的事件称之为槽。

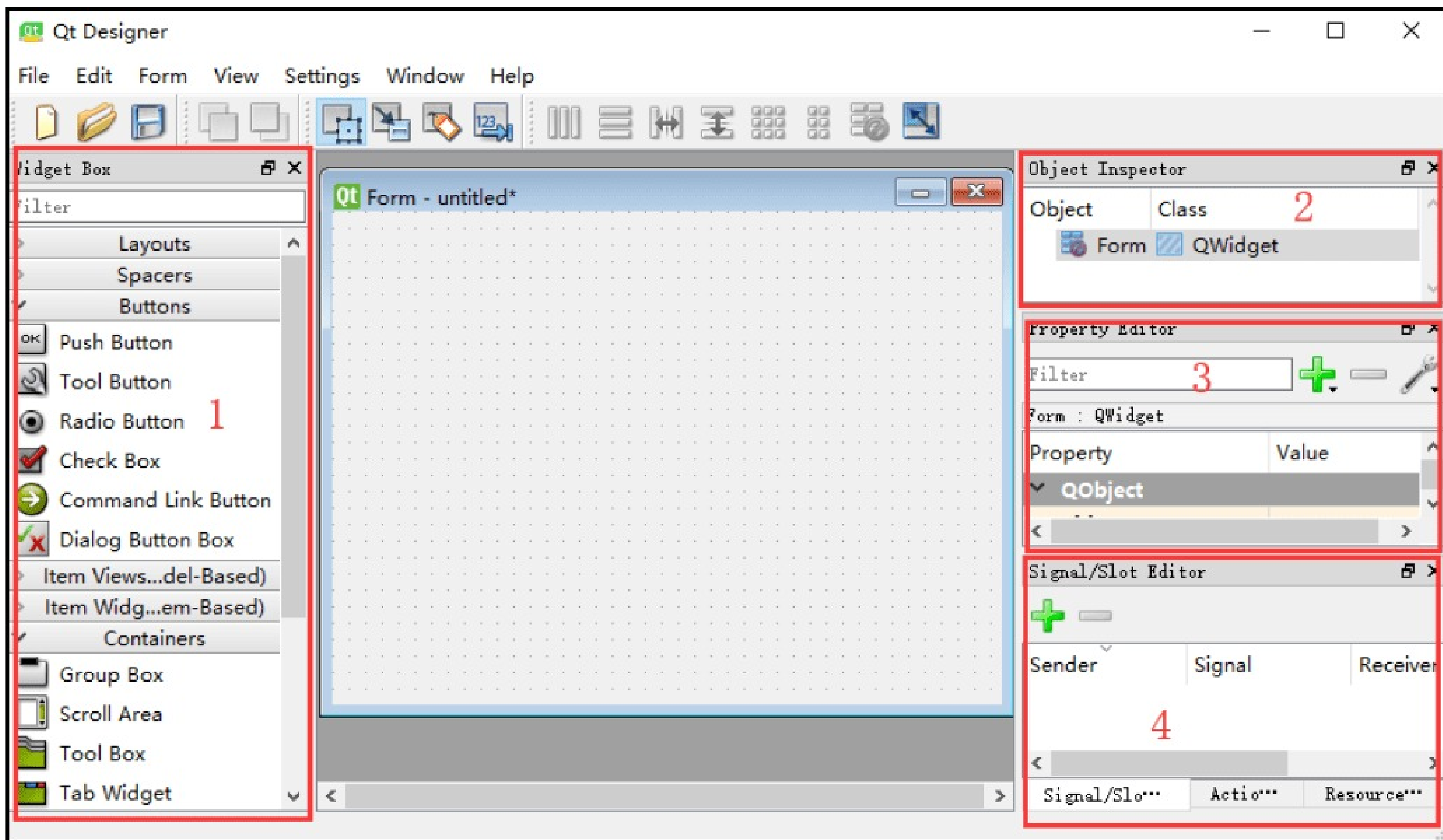


图 21-11　Qt Designer 界面

软件主界面的背景图和三个按钮分别由 QFrame 和 QPushButton 控件实现，在左侧的控件区里拖拉 Frame 和 Push Button 到正中间区域，并对每个控件的 styleSheet 属性进行配置，将文件目录的 ico 文件夹的 ico 图标加载到 QFrame 和 QPushButton 控件，如图 21-12 所示。

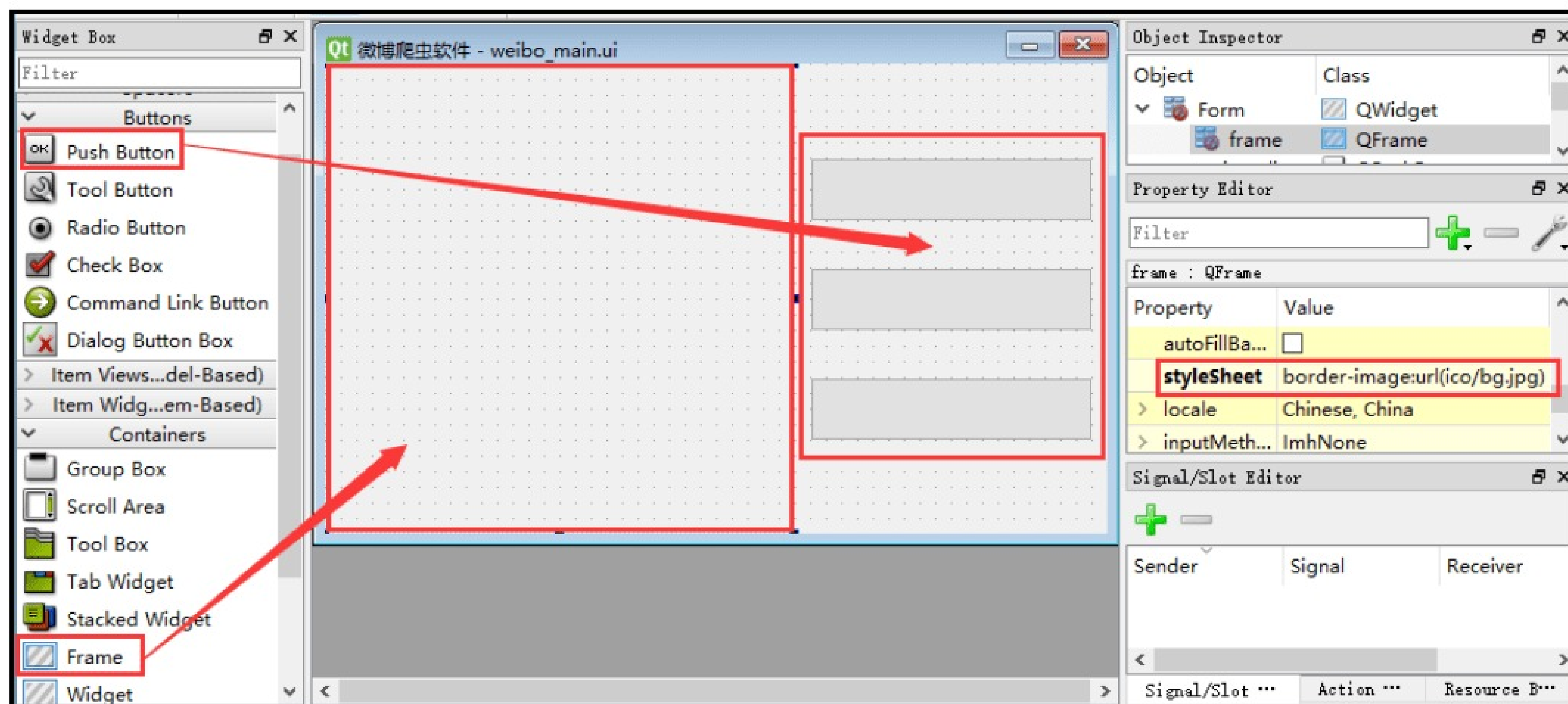


图 21-12　设计软件主界面

控件的 styleSheet 属性值是固定的语法，不同的控件只需改变 ico 的文件名即可，比如 QFrame 控件是放置微博背景图，即项目文件夹 ico 的 bg.jpg 文件。将设计好的界面保存为 weibo_main.ui，该文件就是项目目录的 weibo_main.ui 文件。

在 PyCharm 左侧的文件目录下，选中 weibo_main.ui 文件并右键选择“External Tool”→“PyUIC”，如图 21-13 所示，PyCharm 会自动执行 Python 指令，将 ui 文件转换 py 文件，即项目文件 weibo_main.py。

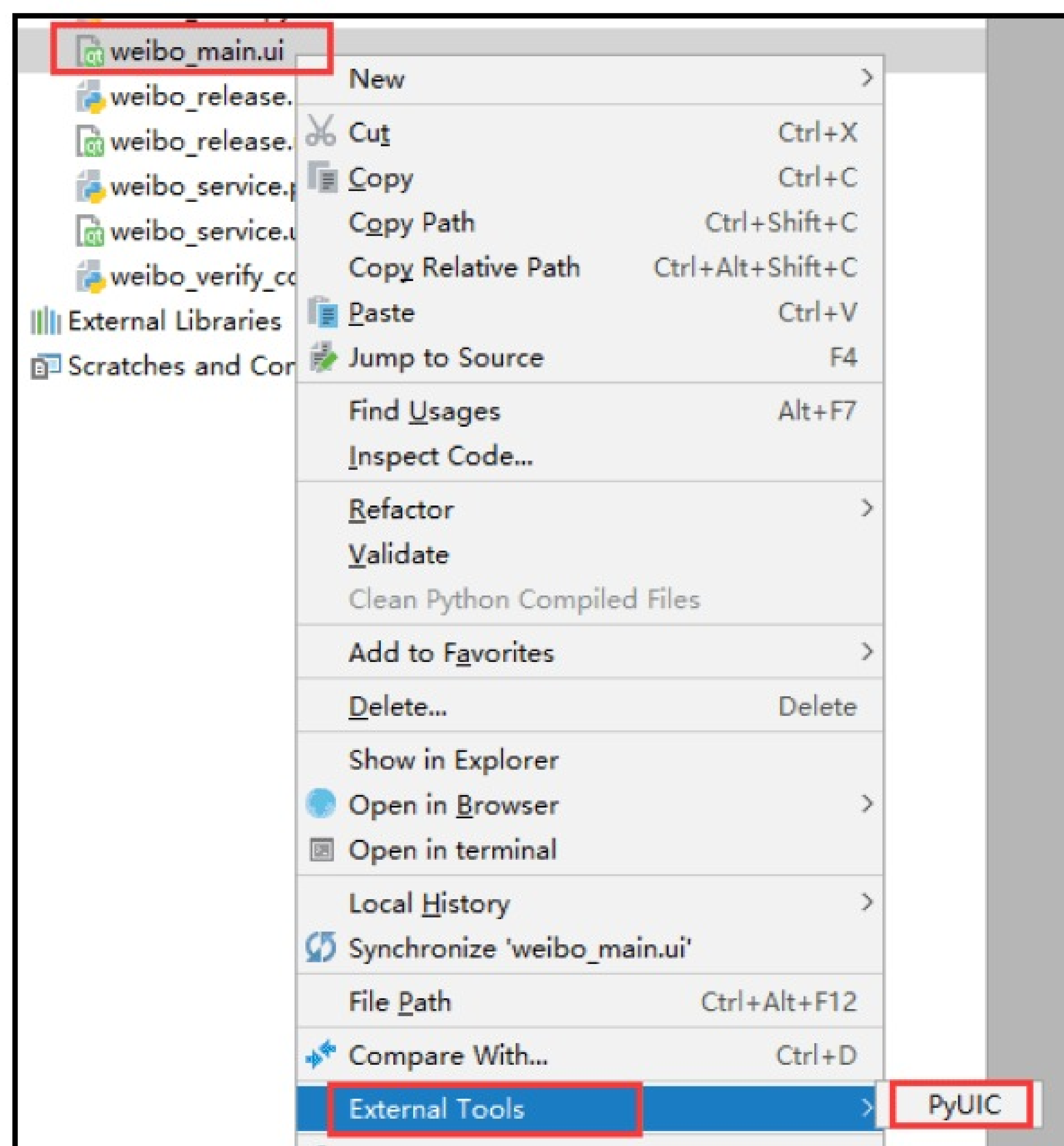


图 21-13　ui 文件转换 py 文件

在 weibo_main.py 文件里定义了 Ui_Form 类，类方法 setupUi 和 retranslateUi 是将 QT Designer 设计的界面以 Python 的代码表示。weibo_main.py 的代码无需修改，若要改动界面设计，只需在

QT Designer 重新设计 weibo_main.ui 文件，然后将 weibo_main.ui 文件转换 weibo_main.py 文件即可。

weibo_main.py 文件是以 Python 的代码来描述软件界面的设计逻辑，如果要将界面呈现出来，还需要编写相应的功能代码，这些功能代码在 main.py 文件里实现，具体的代码如下：

```
from PyQt5 import QtCore, QtGui, QtWidgets
from weibo_main import Ui_Form
from service import weibo_service_logic
from collect import weibo_collect_logic
from release import weibo_release_logic
import sys

# 软件主界面
class main_windows(QtWidgets.QWidget, Ui_Form):
    # 自定义初始化函数
    def __init__(self, parent=None):
        super(main_windows, self).__init__(parent)
        self.setupUi(self)
        # 设置 logo 和图片
        self.setWindowIcon(QtGui.QIcon('ico/logo.png'))
    # 定义界面运行函数
    def show_win(self):
        # setFixedSize 固定界面大小
        self.setFixedSize(self.width(), self.height())
        # 将最大化按钮设为不可用
        self.setWindowFlag(QtCore.Qt.WindowMaximizeButtonHint, False)
        self.show()
    # 定义界面关闭函数
    def close_win(self):
        self.close()

if __name__ == '__main__':
    # 实例化 PyQt5 对象
    app = QtWidgets.QApplication(sys.argv)
    # 实例化软件主界面
    mw = main_windows()
    # 其他功能界面的实例化对象
    service = weibo_service_logic()
    collect = weibo_collect_logic()
    release = weibo_release_logic()

    # 显示软件主界面
    mw.show_win()
    # 软件主界面的按钮绑定功能函数
    mw.main_proxy.clicked.connect(mw.close_win)
    mw.main_proxy.clicked.connect(service.show_win)
    mw.main_collect.clicked.connect(mw.close_win)
    mw.main_collect.clicked.connect(collect.show_win)
    mw.main_release.clicked.connect(mw.close_win)
    mw.main_release.clicked.connect(release.show_win)

    # 其他功能界面的"主菜单"按钮绑定功能函数
    collect.main_win.clicked.connect(collect.close_win)
```

```
        collect.main_win.clicked.connect(mw.show_win)
        release.main_win.clicked.connect(release.close_win)
        release.main_win.clicked.connect(mw.show_win)
        service.main_win.clicked.connect(service.close_win)
        service.main_win.clicked.connect(mw.show_win)
        # 软件结束运行
        sys.exit(app.exec_())
```

整段代码分为两大部分：定义 main_windows 类和设置文件运行入口，具体说明如下：

（1）main_windows 类是继承 weibo_main.py 文件的 Ui_Form 类，使 main_windows 类具有 Ui_Form 类的全部特性，并自定义初始化方法__init__、类方法 show_win 和 close_win，这些方法可进一步完善软件主界面的功能。

（2）设置文件运行入口是通过 if __name__ == '__main__'实现，首先创建 PyQt5 的实例化对象 app 以及各个界面的实例化对象，然后在各个界面的“主菜单”按钮绑定功能函数，比如在主界面单击“采集”按钮就会进入微博采集界面，并关闭软件主界面，这个单击操作涉及了两个功能，分别是运行微博采集界面和关闭主界面，相应的代码及说明如下：

```
    #关闭软件主界面
    #main_collect 是指主界面的采集按钮
    mw.main_collect.clicked.connect(mw.close_win)
    #运行微博采集界面
    #collect.show_win 的 collect 是微博采集界面实例化对象
    #show_win 是该对象下定义的函数
    mw.main_collect.clicked.connect(collect.show_win)
```

21.4 相关服务界面

从相关服务界面的效果图看到，界面分为打码服务和代理服务，这些服务都有第三方网站提供，要使用这些服务，只需在界面上设置相关的账号信息即可。对于这些服务的使用，本书不做详细介绍，读者可单击软件的“购买打码服务”和“购买代理服务”按钮，进入官网了解使用方法。

相关服务界面的设计共有 7 个 QPushButton 控件、3 个 QLineEdit 以及其他布局控件，各个控件所实现的功能说明如下：

- 主菜单。关闭相关服务界面并运行软件主界面，实现界面之间的切换。
- 购买打码服务。单击该按钮即可在本地的浏览器打开第三方打码平台的官网。
- 购买代理服务。单击该按钮即可在本地的浏览器打开第三方代理 IP 平台的官网。
- 账号和密码：输入打码平台的用户账号和密码，根据账号密码调用 API 接口来获取验证码识别服务。
- 单号。输入代理 IP 平台的所提供的单号，根据单号调用 API 接口来获取代理 IP。
- 验证。分别验证账号密码（单号）是否正确，并将账号密码（单号）写入配置文件，当软件下次运行时，无需再次设置。
- 清空。清空软件里所填写的账号密码（单号）及配置文件的信息。

界面的设计由 weibo_service.ui 和 weibo_service.py 实现，软件功能由 service.py 实现。其中界面设计文件 weibo_service.py 是由 weibo_service.ui 转换而成，而 weibo_service.ui 是 QT Designer 的设计文件，大致的设计流程与软件主界面的一致，界面设计如图 21-14 所示。

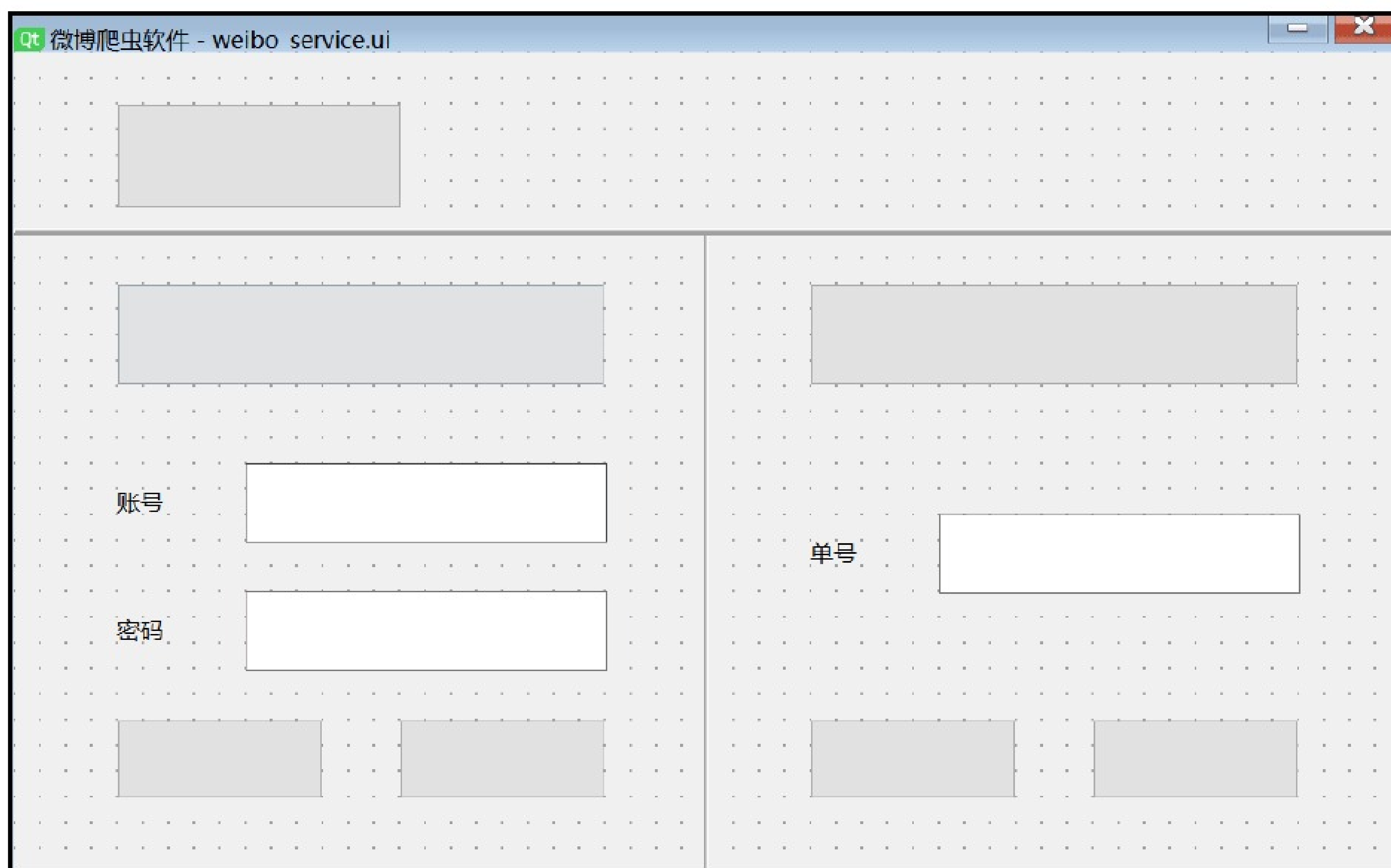


图 21-14　相关服务界面

图上的按钮设置了按钮属性 objectName 和 styleSheet，objectName 是对控件进行命名，在代码里，操控按钮由按钮的命名实现，比如设置按钮的信号槽（触发事件）；styleSheet 是设置按钮的 ico 图标，设置效果如图 21-2 所示。至于界面的其他设计，读者可以在 QT Designer 打开 weibo_service.ui 了解更多。

界面的功能由 service.py 文件实现，同时该文件也是界面的运行文件。文件的代码结构与 main.py 相似，定义软件功能类和设置文件运行入口，具体的代码如下：

```
from PyQt5 import QtCore, QtGui, QtWidgets
from weibo_service import Ui_Dialog
import requests
import configparser
import os
import sys

# 相关服务的功能逻辑
class weibo_service_logic(QtWidgets.QWidget, Ui_Dialog):
    # 重写初始化函数
    def __init__(self, parent=None):
        super(weibo_service_logic, self).__init__(parent)
        self.setupUi(self)
        # 设置左上方的 logo
        self.setWindowIcon(QtGui.QIcon('ico/logo.png'))
        # 设置按钮'购买打码服务'的功能
        self.buy_code.clicked.connect(self.buy_code_def)
        # 设置按钮'购买代理服务'的功能
        self.buy_proxy.clicked.connect(self.buy_proxy_def)
```

```
        # 设置打码服务的按钮'验证'的功能
        self.code_bt.clicked.connect(self.set_code)
        # 设置代理服务的按钮'验证'的功能
        self.proxy_bt.clicked.connect(self.set_proxy)
        # 设置打码服务的按钮'清空'的功能
        self.code_clean.clicked.connect(self.code_clean_def)
        # 设置代理服务的按钮'清空'的功能
        self.proxy_clean.clicked.connect(self.proxy_clean_def)

        # 读取配置文件，软件再次运行无需重复设置
        conf = configparser.ConfigParser()
        if os.path.exists('./temp/conf.ini'):
            conf.read('./temp/conf.ini')
            if 'config' in conf.keys():
                temp = conf['config']
                if 'proxies' in temp.keys():
                    self.proxy_text.setText(conf['config']['proxies'])
                if 'user' in temp.keys():
                    self.code_account.setText(conf['config']['user'])
                if 'password' in temp.keys():
                    self.code_password.setText(conf['config']['password'])

    # 购买打码服务
    def buy_code_def(self):
        QtGui.QDesktopServices.openUrl(QtCore.QUrl(
                                     'http://www.yundama.com/'))

    # 购买代理 IP 服务
    def buy_proxy_def(self):
        QtGui.QDesktopServices.openUrl(QtCore.QUrl(
                                     'http://www.data5u.com/'))

    # 验证单号
    def set_proxy(self):
        proxy_key = self.proxy_text.text().strip()
        if proxy_key:
            # 获取代理 IP
            url = 'http://api.ip.data5u.com/dynamic/get.html?
                    order='+proxy_key + '&random=true&sep=5'
            r = requests.get(url)
            # 判断 IP 代理是否过期
            if 'success' in str(r.text):
                warm_info = '单号未充值或者单号已经到期'
            else:
                warm_info = '验证成功'
                # 写入配置文件
                conf = configparser.ConfigParser()
                if os.path.exists('./temp/conf.ini'):
                    conf.read('./temp/conf.ini')
                else:
                    conf.add_section('config')
                conf.set('config', 'proxies', proxy_key)
                conf.write(open('./temp/conf.ini', 'w'))
        else:
```

```
            warm_info = '请输入单号'
        self.proxy status.setText(warm info)

    # 验证打码服务
    def set code(self):
        username = self.code account.text().strip()
        password = self.code password.text().strip()
        url = 'http://api.yundama.com/api.php?method=balance'
        data = {'username': username, 'password': password,
                'appkey': 'c5e26d1a207df586d7aaec21522dd446',
                'appid': '4055'}
        r = requests.post(url, data=data)
        # 判断账号是否正常
        if r.json()['ret'] == 0:
            # 写入配置文件
            conf = configparser.ConfigParser()
            if os.path.exists('./temp/conf.ini'):
                conf.read('./temp/conf.ini')
            else:
                conf.add section('config')
            conf.set('config', 'user', username)
            conf.set('config', 'password', password)
            conf.write(open('./temp/conf.ini', 'w'))
            self.proxy status.setText('验证成功,余额为
                                      '+str(r.json()['balance']))
        elif r.json()['ret'] == -1001:
            self.proxy status.setText('打码平台账号密码错误')
        elif r.json()['ret'] == -1007:
            self.proxy status.setText('打码平台余额为 0，请及时充值')

    # 清空单号设置
    def proxy clean def(self):
        conf = configparser.ConfigParser()
        if os.path.exists('./temp/conf.ini'):
            conf.read('./temp/conf.ini')
            conf.set('config', 'proxies', '')
            conf.write(open('./temp/conf.ini', 'w'))
        self.proxy_status.setText('已清空单号')
        self.proxy text.setText('')

    # 清空打码设置
    def code clean def(self):
        conf = configparser.ConfigParser()
        if os.path.exists('./temp/conf.ini'):
            conf.read('./temp/conf.ini')
            conf.set('config', 'yunmauser', '')
            conf.set('config', 'yunmapassword', '')
            conf.write(open('./temp/conf.ini', 'w'))
        self.proxy_status.setText('已清空打码平台账号密码')
        self.code account.setText('')
        self.code password.setText('')

    # 运行界面
    def show_win(self):
```

```
            self.setFixedSize(self.width(), self.height())
            self.setWindowFlag(QtCore.Qt.WindowMaximizeButtonHint,False)
            self.show()
        # 关闭界面
        def close_win(self):
            self.close()

    # 文件运行入口
    if __name__ == '__main__':
        app = QtWidgets.QApplication(sys.argv)
        ex = weibo_service_logic()
        ex.show()
        sys.exit(app.exec_())
```

上述代码定义了 weibo_service_logic 类，这是相关服务界面的功能类。它定义了多个类方法，这些方法是实现界面的功能，具体说明如下：

（1）__init__()用于重写初始化函数，为界面上的按钮绑定相应的功能函数。

（2）buy_code_def()在初始化函数绑定了“购买打码服务”按钮，实现浏览器访问打码平台的官网。

（3）buy_proxy_def()在初始化函数绑定了“购买代理服务”按钮，实现浏览器访问代理平台的官网。

（4）set_proxy()在初始化函数绑定了代理的“验证”按钮，根据文本框的单号与第三方平台的 AIP 接口进行验证，若验证成功，则写入 temp 文件夹的配置文件 conf.ini，否则提示验证失败信息。

（5）set_code()在初始化函数绑定了打码的“验证”按钮，根据文本框的账号密码与第三方平台的 API 接口进行验证，若验证成功，则写入 temp 文件夹的配置文件 conf.ini，否则提示验证失败信息。

（6）proxy_clean_def()在初始化函数中绑定了代理的“清空”按钮，实现文本框和配置文件的清空。

（7）code_clean_def()在初始化函数中绑定了打码的“清空”按钮，实现文本框和配置文件的清空。

（8）show_win()运行相关服务界面，用于 main.py 文件的文件运行入口。

（9）close_win()关闭相关服务界面，用于 main.py 文件的文件运行入口。

在 service.py 设置文件运行入口是为让相关服务界面单独运行，这样可方便开发人员测试软件功能是否正常。如果单独运行相关服务界面，软件的“主菜单”按钮是没有界面切换功能的，因为该按钮的功能是在 main.py 文件的文件运行入口设置。

21.5 微博采集界面

微博采集界面是将热门微博的爬虫与软件开发相结合，这是爬虫软件的核心开发思想，使用者在软件设置中的信息会以函数参数的形式传递给爬虫函数，爬虫根据用户设置的信息去执行相应

的爬取操作。

微博采集界面使用了多个不同类型的控件，使用 QT Designer 打开 weibo_collect.ui，进一步分析软件界面设计原理，如图 21-15 所示。

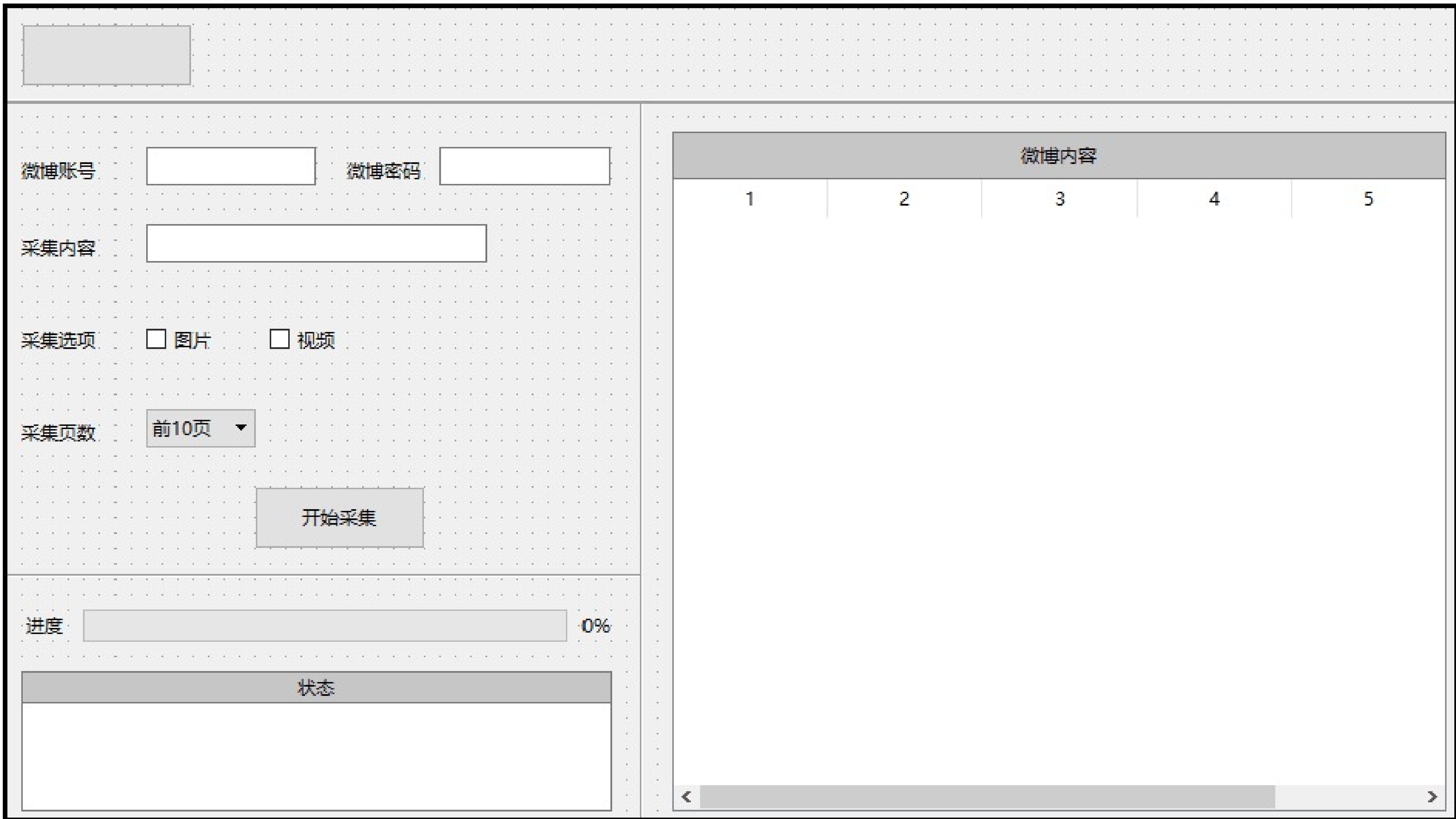


图 21-15　微博采集界面

微博采集界面共有 9 个功能控件以及多个优化控件，优化控件是为了美化界面，实质上并没有太多的功能，比如 Line 和 QLabel 控件等。界面的功能控件说明如下：

- 主菜单。返回软件主界面，并关闭微博采集界面。
- 账号密码。设置微博采集的账号，因为爬取热门微博需要登录微博用户，所以采集微博之前，需要根据文本框的内容进行微博登录。
- 采集内容。采集热门微博需根据关键词筛选相关微博再进行爬取，而该文本框是为爬虫提供动态的关键词。
- 采集选项。默认情况下只爬取微博的文字内容，如需爬取微博的图片和视频，可勾选相应的选项。
- 采集页数。微博搜索只提供 50 页的相关微博，通过设置页数来控制爬取范围。
- 采集按钮。用于启动或暂停爬虫的运行，爬取结果存储在 CSV 文件中。
- 进度条。将采集页数进行分段爬取，每完成一次爬取就会显示相应的进度。
- 状态。显示爬虫的信息提示，如用户登录是否成功、微博采集是否完成等信息。
- 微博内容。读取 CVS 文件，将爬取结果以数据表格形式呈现。

界面的设计流程与软件主界面相同，将界面设计文件 weibo_collect.ui 转换成 weibo_collect.py 文件，然后在 collect.py 中定义子类 weibo_collect_logic，通过继承并重写 weibo_collect.py 的 Ui_Dialog 类。子类重写了初始化函数并自定义了 5 个类方法，具体的代码如下：

```
from PyQt5 import QtCore, QtGui, QtWidgets
from weibo_collect import Ui_Dialog
from weibo import *
from PyQt5.QtCore import QBasicTimer
```

```
import csv, time, os, configparser, sys

# 热门微博采集
class weibo_collect_logic(QtWidgets.QWidget, Ui_Dialog):
    def __init__(self, parent=None):
        super(weibo_collect_logic, self).__init__(parent)
        self.setupUi(self)
        # 设置左上方的 logo
        self.setWindowIcon(QtGui.QIcon('ico/logo.png'))
        # 设置表格的表头格式
        self.collect_data.setHorizontalHeaderLabels(['用户','文本内容','图片',
                                                     '视频', '采集日期'])
        self.collect_data.verticalHeader().setStyleSheet(
                       "QHeaderView::section {background:rgb(230, 230, 230)}")
        self.collect_data.horizontalHeader().setStyleSheet(
                       "QHeaderView::section {background:rgb(230, 230, 230)}")
        # 设置按钮'采集'的功能
        self.collect_start.clicked.connect(self.collect_weibo_data)
        # 定义进度条对象
        self.timer = QBasicTimer()
        self.step = 0
        # 设置属性，在函数之间调用
        self.session = ''
        self.keyword = ''
        self.pagenumber = 1
        # 读取配置文件
        conf = configparser.ConfigParser()
        if os.path.exists('./temp/conf.ini'):
            conf.read('./temp/conf.ini')
            if 'config' in conf.keys():
                temp = conf['config']
                if 'collect_username' in temp.keys():
                    self.collect_user.setText(conf['config']
                                              ['collect_username'])
                if 'collect_password' in temp.keys():
                    self.collect_password.setText(conf['config']
                                                  ['collect_password'])

    # 进度条
    def timerEvent(self, event):
        # 进度条已满，即完成采集，执行初始化，为下次采集准备
        if self.step >= 100:
            self.timer.stop()
            self.collect_state.setPlainText('采集完成' + '\n'+
                                            self.collect_state.toPlainText())
            self.collect_start.setText('开始采集')
            self.step = 0
            self.session = ''
            self.pagenumber = 1
            self.write_table()
            return
        # 调用函数 collect_weibo，采集微博
        collect_weibo(keyword=self.keyword, session=self.session,
                      pagenumber=self.pagenumber, proxies={},
```

```
                    get img=self.get img, get video=self.get video)
        # 爬取完成后，设置进度条
        self.pagenumber += 1
        self.step = self.step + self.speed
        self.progressBar.setValue(self.step)
        time.sleep(2)

    # 绑定'开始采集'按钮的功能函数
    def collect weibo data(self):
        if self.step == 0:
            # 获取采集选项
            self.get img = self.select pic.isChecked()
            self.get video = self.select video.isChecked()
            self.keyword = self.collect keyword.text().strip()

            # 清空 datatable 数据
            self.collect_data.setRowCount(0)
            # 获取登录账号密码
            username = self.collect user.text().strip()
            password = self.collect password.text().strip()
            # 登录
            if username and password and self.keyword:
                # 登录验证
                login info = login(username, password)
                # 判断验证结果
                if 'session' in login info.keys():
                    self.session = login info['session']
                    # 写入配置文件
                    conf = configparser.ConfigParser()
                    if os.path.exists('./temp/conf.ini'):
                        conf.read('./temp/conf.ini')
                    else:
                        conf.add section('config')
                    conf.set('config', 'collect username', username)
                    conf.set('config', 'collect_password', password)
                    conf.write(open('./temp/conf.ini', 'w'))

            # 根据爬取页数进行分段处理
            if self.collect page.currentIndex() == 0:
                pagenumber = 10
            elif self.collect page.currentIndex() == 1:
                pagenumber = 20
            else:
                pagenumber = 50
            self.speed = 100 / pagenumber

        # 暂停或开始微博采集
        if self.timer.isActive():
            self.timer.stop()
            self.collect start.setText('继续采集')
        elif self.timer.isActive()==False and self.session and self.keyword:
            self.timer.start(100, self)
            self.collect start.setText('暂停采集')
        # 判断关键词是否为空
```

```
            elif self.keyword == '':
                self.collect_state.setPlainText('请输入关键词' +
                                    '\n' + self.collect_state.toPlainText())
            # 判断当前微博是否已登录
            elif self.session == '':
                self.collect_state.setPlainText('微博账号或密码错误' +
                                    '\n' + self.collect_state.toPlainText())
        # 读取 CSV 文件，将文件内容写入 Table
        def write_table(self):
            if os.path.exists('./temp/data.csv'):
                csv_reader = csv.reader(open('./temp/data.csv',
                                     'r', encoding='gb18030'))
                for index, row in enumerate(iter(csv_reader)):
                    if index != 0:
                        self.collect_data.setRowCount(
                                      self.collect_data.rowCount() + 1)
                        rownumber = self.collect_data.rowCount()
                        for i in range(5):
                            newItem = QtWidgets.QTableWidgetItem(row[i])
                            self.collect_data.setItem(rownumber - 1, i, newItem)
                self.collect_data.sortByColumn(6, QtCore.Qt.DescendingOrder)

        # 定义界面运行函数
        def show_win(self):
            self.setFixedSize(self.width(), self.height())
            self.setWindowFlag(QtCore.Qt.WindowMaximizeButtonHint, False)
            self.show()
        # 定义界面的关闭函数
        def close_win(self):
            self.close()

    # 软件的运行入口
    if __name__ == '__main__':
        app = QtWidgets.QApplication(sys.argv)
        ex = weibo_collect_logic()
        ex.show_win()
        sys.exit(app.exec_())
```

weibo_collect_logic 类重写初始化函数__init__并定义类方法 timerEvent()、collect_weibo_data()、write_table()、show_win()和 close_win()，分别说明如下：

（1）初始化函数__init__()分别设置了软件界面左上方的 ico 图标、数据表格的表头内容和样式、“开始采集”按钮绑定功能函数、定义进度条对象、定义类属性和读取配置文件，大致的说明如下：

- 初始化函数的作用是为软件界面的运行提供基础设置，定义类属性是为方便函数之间的使用，可以简单理解为类的全局变量。
- 读取配置文件是将配置文件里的微博账号和密码自动填写到软件的文本框，这样无需使用者重复填写。
- timerEvent()是进度条对象特定的方法，由于爬取方式是以进度条为主，而进度条的执行由类方法 timerEvent()实现，timerEvent()的逻辑如下：

- 首先判断类属性 step 是否大于或等于 100，当类属性 step 符合条件，说明进度条已满，即完成微博采集，因此将某些类属性进行初始化，为下次的微博采集作准备。
- 然后调用爬虫函数 collect_weibo()，实现采集微博，函数参数以类属性传入，这些类属性都由类方法 collect_weibo_data()设置。
- 最后修改进度条的相关参数，代表本次进度的执行已完成。

（2）collect_weibo_data()是绑定“开始采集”按钮的功能函数，该方法用于读取控件上的数据，并根据这些数据配置相关属性，用于类方法 timerEvent()的使用；此外还能控制进度的运行、暂停以及控件内容的判断，说明如下：

- 首先判断类属性 step 是否等于 0，若为 0，说明进度条是第一次执行，因此对软件的控件内容进行读取并登录微博账号。
- 如果登录成功，将文本框的微博账号密码写入配置文件，方便下次读取；此外还根据爬取的页数来计算进度条的运行次数，类属性 speed 是每次进度条的长度，比如爬取前 20 页的微博，那么每次进度条的长度为 5，而进度条的总数为 100，每爬取一页，进度条的长度都会累加 5，直到进度条满 100 为止。
- 然后判断进度条对象 timer 是否处于活动状态，如果处于活动状态，当再次单击“采集”按钮的时候，对象 timer 就会暂停运行，再次单击“采集”按钮就会继续运行爬虫，这是一种运行状态的切换。当对象 timer 在运行的时候，它就会自动调用类方法 timerEvent()，从而执行爬虫程序。
- 最后对关键词内容和用户登录状态进行判断，这些判断信息都会显示在状态文本框中。

（3）write_table()方法用于将 CSV 文件内容写入数据表格。由于爬取结果是写入 CSV 文件，因此爬虫运行完成后，软件就会自动读取 CSV 文件，将爬取结果写在软件的数据表格，方便使用者查看。在类方法 timerEvent()里，当类属性 step 大于或等于 100 的时候，它除了将类属性重置之外，还调用了类方法 write_table()，这保证爬虫运行完成后，软件可自动读取 CSV 文件并加载到数据表格。

（4）类方法 show_win()和 close_win()的作用与相关服务界面的作用一致，此处就不再赘述。

总的来说，整个 collect.py 文件的核心代码是类方法 timerEvent()和 collect_weibo_data()，两者之间相辅相成。在 collect_weibo_data()里，通过对象 timer 的状态来决定是否运行 timerEvent()方法，同时 timerEvent()所使用的数据都是以类属性表示，这些类属性由 collect_weibo_data()方法定义和赋值，也就是说，两个方法之间通过类属性来实现数据交互。

21.6　微博发布界面

微博发布与微博采集界面在设计上有一定的相似之处，但两者在功能上存在着明显的差异，主要体现在以下几点：

（1）数据表格具有编辑功能，如自动增加行数、整行删除、颜色填充等功能，而微博采集只具有数据查看功能。

（2）新增图片添加和定时功能，前者是打开系统的文件对话框；后者是由 QDateEdit 和 QCombobox 控件实现。

（3）不同账户的微博批量发布涉及到代理 IP 的验证与使用，微博定时发布的时间验证与设置。

上述是微博发布的主要功能，在界面设计上，打开 weibo_release.ui 可以看到软件的功能控件主要有 QPushButton、QTableWidget、QProgressBar、QDateEdit 和 QCombobox 以及其他界面优化控件等，如图 21-16 所示。

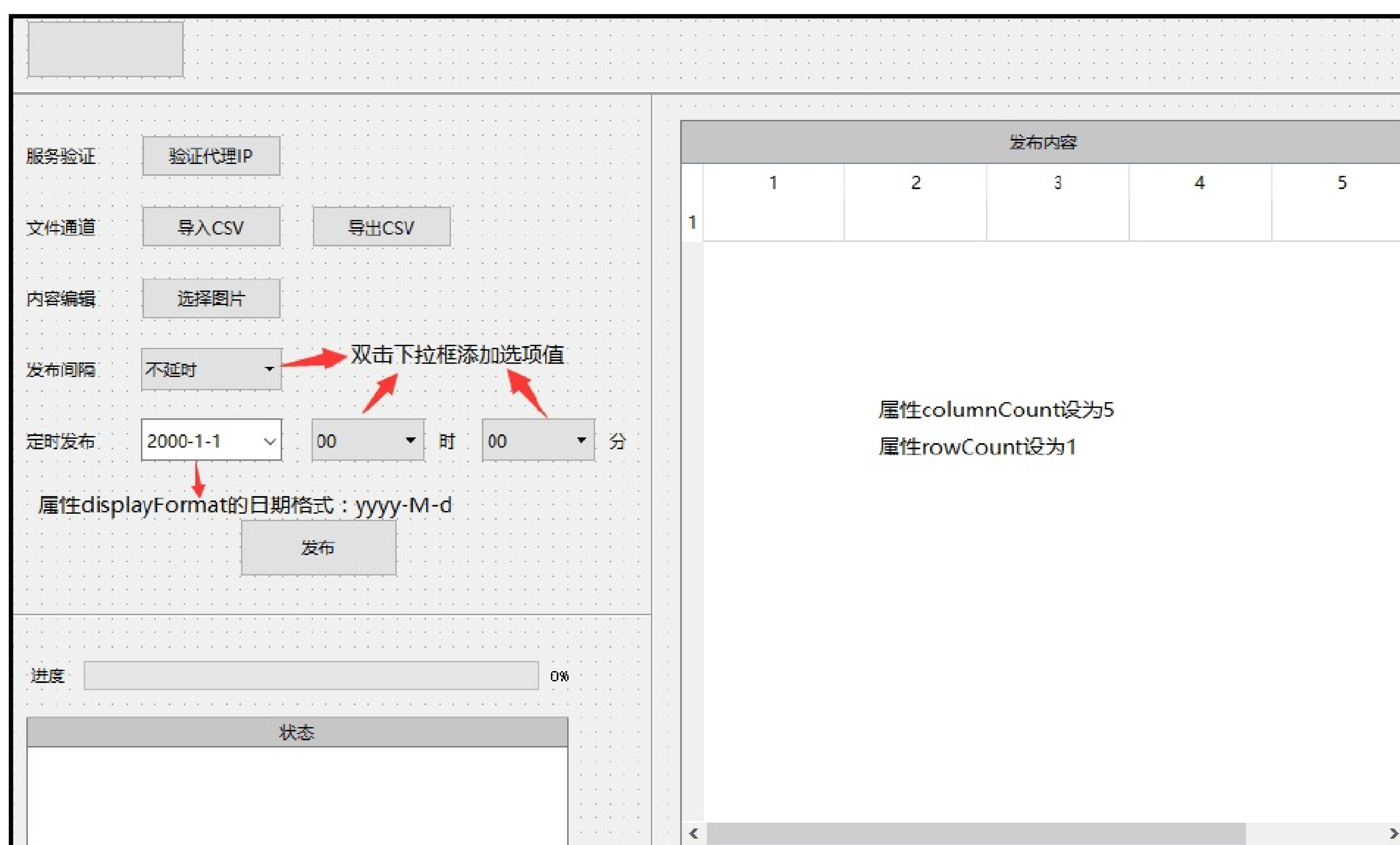


图 21-16　微博发布界面

从图 21-16 上看到，在 QT Designer 里，双击下拉框（QCombobox）可以设置选项值；日期选项框（QDateEdit）用于设置属性 displayFormat 的日期格式；右侧的数据表格 QTableWidget 可分别设置 columnCount 和 rowCount，代表表格的列数和行数，这些属性设置可用于软件的功能开发。此外，每个功能控件的说明如下：

- 主菜单。返回软件主界面，并关闭微博发布界面。
- 服务验证。从配置文件 conf.ini 读取代理 IP 的单号，根据单号调用第三方平台 API 接口获取代理 IP 的 IP 地址。
- 文件通道。将数据表格的数据导出到 CSV 文件或者将 CSV 文件的数据导入到数据表格。
- 内容编辑。提供图片上存功能，以图片文件夹为单位，比如发送某条微博，该微博附带 9 张图片，则在数据表格选中该微博所在的行数，然后单击“选择图片”，选择图片所在的文件夹，在数据表格的“图片”将会出现图片的路径地址。
- 发布间隔。设有 4 个选项：不延时、延时 5 秒、延时 10 秒和延时 15 秒，用于设置每条微博的发送间隔，如果不使用代理 IP 而且同一个账号发送多条微博，没有延时的情况下很容易将微博服务器判为爬虫程序，因此设置每条微博的发送延时可以巧妙地避开反爬虫机制。
- 定时发布。由一个 QDateEdit 和两个 QCombobox 控件实现日期设置，可以设置每条微博

的定时发布功能。QDateEdit 是设置日期的年月日；QCombobox 是设置日期的时和分。当三个控件的内容发生变化的时候，数据表格的“定时发布”也会随之变化。

- 发布。触发微博的发布功能，将数据表格的每行数据逐一执行，并将执行结果分别返回到状态文本框以及在数据表格填充相应的颜色。
- 进度条。将发送微博的数量进行分段发布，每个进度代表一条微博。
- 状态。显示爬虫的信息提示，如用户登录是否成功，微博发布是否完成等信息。
- 发布内容。用于编辑待发送微博的内容及相关设置。为了方便使用者编辑微博内容，数据表格设有自动新增行数，根据最后一行的“账号”是否为空来决定是否新增行数；还可以设置键盘事件，选中某行数据并按“Delete”键即可删除当前选中行；此外还设有表格颜色的填充功能，根据微博发布的执行结果来填充相应的颜色。

根据每个控件的功能描述，整个 release.py 的代码如下：

```
from PyQt5 import QtCore, QtGui, QtWidgets
from weibo release import Ui Dialog
from weibo import *
from PyQt5.QtCore import QBasicTimer
import csv, time, datetime
import os, configparser
import requests
import sys

# 微博发布
class weibo release logic(QtWidgets.QWidget, Ui Dialog):
    def   init  (self, parent=None):
        super(weibo release logic, self).  init  (parent)
        self.setupUi(self)
        # 设置 logo
        self.setWindowIcon(QtGui.QIcon('ico/logo.png'))
        # 设置数据表格
        self.release table.setHorizontalHeaderLabels(
                            ['账号', '密码', '内容', '图片', '定时发布'])
        self.release table.setRowCount(1)
        # 表格绑定 tableset，让表格自动添加行数
        self.release table.currentCellChanged[
                      'int', 'int', 'int', 'int'].connect(self.tableset)
        # 设置表格的表头
        self.release table.verticalHeader().setStyleSheet(
                  "QHeaderView::section {background:rgb(230, 230, 230)}")
        self.release_table.horizontalHeader().setStyleSheet(
                  "QHeaderView::section {background:rgb(230, 230, 230)}")

        # 导入 CSV 的数据
        self.importcsv.clicked.connect(self.importcsv def)
        # 导出到 CSV
        self.exportcsv.clicked.connect(self.exportcsv def)
        # 图片按钮绑定功能函数
        self.release pic.clicked.connect(self.showDialog)
        # 发布按钮绑定功能函数
        self.release_bt.clicked.connect(self.timer_and_weibo)
```

```
        # 验证代理按钮绑定功能函数
        self.user check proxy.clicked.connect(self.check proxy)
        # 数据表格设定键盘事件
        self.release table.activated['QModelIndex'].
                                connect(self.keyPressEvent)

        # 定时发布设置
        now = datetime.datetime.now()
        now_year = now.strftime('%Y')
        now month = now.strftime('%m')
        now day = now.strftime('%d')
        now hour = now.strftime('%H')
        now minute = now.strftime('%M')
        self.dateEdit.setDate(QtCore.QDate(
                        int(now year),int(now month),int(now day)))
        # 将定时发送的时间设置为当前时间
        self.hour.setCurrentText(now_hour)
        self.minute.setCurrentText(now minute)
        # 当时间控件发生改变而触发的方法，判断设定的时间是否符合微博的延时发送
        self.hour.currentIndexChanged['int'].connect(self.set time)
        self.minute.currentIndexChanged['int'].connect(self.set time)
        self.dateEdit.dateChanged['QDate'].connect(self.set time)
        # 初始化类属性
        self.pic list = []
        self.timer = QBasicTimer()
        self.step = 0
        self.index = 0
        self.row number list = []
        self.session dict = {}

    # 定义键盘事件，用于快速删除数据表格的整行数据
    def keyPressEvent(self, e):
        keyEvent = QtGui.QKeyEvent(e)
        getrow = self.release table.currentRow()
        if keyEvent.key() == QtCore.Qt.Key Delete:
            self.release table.removeRow(getrow)
            self.tableset()

    # 发博微博
    def release weibo(self):
        # 验证用户和发布微博,先判断用户登录状态
        self.check proxy()
        # 登录验证用户
        username = self.release table.item(
                    self.row number list[self.index], 0).text().strip()
        password = self.release table.item(
                    self.row number list[self.index], 1).text().strip()
        # 判断是否已登录过
        if username in self.session dict.keys() and self.proxies=={}:
            user = self.session dict[username]
            time.sleep(3)
        else:
            user = login(username,password,proxies=self.proxies)
        # 登录成功
```

```
if user['code'] == '1000':
    # 写入 session_dict
    self.session_dict[username] = user
    # 获取登录信息
    session = user['session']
    person_info = user['info']
    location = person_info['location']
    watermark = person_info['watermark']
    nick = person_info['nick']
    # 判断是否有图片
    pic = self.release_table.item(
           self.row_number_list[self.index], 3)
    if pic:
        if pic.text():
            for i in pic.text().split(','):
                i = i.replace(r'\\', '')
                if os.path.exists(i):
                    self.pic_list.append(i)
    # 设置函数参数
    self.value = self.release_table.item(
                  self.row_number_list[self.index], 2).text()
    try:
        self.addtime = self.release_table.item(
                         self.row_number_list[self.index], 4).text()
    except:
        self.addtime = ''
    # 上存图片
    if self.pic_list:
        pic_list = upload_pic(watermark, nick, session,
                     file_list=self.pic_list,proxies=self.proxies)
        send_type = 'pic'
        self.pic_list = []
    else:
        pic_list = []
        send_type = 'words'
    # 发送微博
    send_status = send_weibo(watermark, location, self.value,
                   session, self.addtime, pic_id_list=pic_list,
                   send_type=send_type, proxies=self.proxies)
    if send_status:
        status_info = '用户：'+username+' 微博发布成功'+'\n'
        # 设置当前表格所在行的颜色，绿色
        self.fill_rpg(127, 255, 170)
        self.success_number += 1
    else:
        status_info = '用户：'+username+' 微博发布失败'+'\n'
        self.fail_number += 1
    # 写入状态信息
    self.state_value.setPlainText(status_info+
                   self.warm_info+self.state_value.toPlainText())
# 登录失败
elif user['code'] == '1001':
    self.state_value.setPlainText(
        '用户：' + username + ' 登录失败，失败原因：
```

```
                    微博账号密码错误或者验证码识别错误' + '\n'+
                    self.state value.toPlainText())
                # 设置当前表格所在行的颜色，黄色
                self.fill rpg(255, 255, 0)
                self.fail number += 1
            elif user['code'] == '1002':
                self.state_value.setPlainText('用户：'+username+
                            ' 登录失败,失败原因：验证码账号密码错误或者余额不足'
                            +'\n' + self.state value.toPlainText())
                # 设置当前表格所在行的颜色，棕色
                self.fill rpg(244, 164, 96)
                self.fail number += 1
            self.index += 1

    # 定义进度条
    def timerEvent(self, event):
        if self.step >= 100:
            self.timer.stop()
            self.release_bt.setText('发布')
            self.state value.setPlainText(
                        '发布完成：成功 ' + str(self.success number)+
                        ' 个，失败 ' + str(self.fail_number) + ' 个'+
                        '\n' + self.state value.toPlainText())
            self.step = 0
            self.row number list = []
            self.index = 0
            self.proxies = {}
            self.warm info = ''
            self.pic list = []
            self.value = ''
            return
        # 发博微博
        self.release weibo()
        # 延时，等待下一个
        self.step = self.step + self.speed
        time.sleep(self.time delay)
        self.progressBar.setValue(self.step)

    # 进度条、按钮和微博发布结合使用
    def timer_and_weibo(self):
        if self.step == 0:
            # 初始化
            # 初始化软件要求
            self.step = 0
            self.speed = 0
            # 计算发布的延时时间
            self.time delay=(self.release delay.currentIndex()*5)
            # 统计个数
            self.success number = 0
            self.fail_number = 0
            # 遍历数据表格的每一行，判断每行的数据是否合理
            rownumber = self.release table.rowCount()
            for i in range(rownumber):
                fill_color = True
```

```
            # 判断表格（账号、密码和内容）是否为空
            username = self.release_table.item(i, 0)
            password = self.release_table.item(i, 1)
            content = self.release_table.item(i, 2)
            # 微博内容的长度不能超过 2000
            if username.text() and password.text()
                                    and content.text()
                                    and len(content.text()) <= 2000:
                self.row_number_list.append(i)
                fill_color = False
            # 根据 fill_color 结果判断是否需要设置颜色
            for k in range(5):
                if self.release_table.item(i, k):
                    value = self.release_table.item(i, k).text()
                else:
                    value = ""
                newItem = QtWidgets.QTableWidgetItem(value)
                if fill_color:
                    if i in self.row_number_list:
                        self.row_number_list.remove(i)
                    newItem.setBackground(QtGui.QColor(200,111,100))
                self.release_table.setItem(i, k, newItem)
        # 去除重复的内容
        self.row_number_list = sorted(set(self.row_number_list))
        # 获取微博发布的用户数，计算进度条的间距
        pagenumber = len(self.row_number_list) if
                        len(self.row_number_list) else 1
        self.speed = 100 / pagenumber

    # 暂停与开始
    if self.timer.isActive():
        self.timer.stop()
        self.release_bt.setText('继续发布')
    elif self.timer.isActive()==False and self.row_number_list:
        self.timer.start(100, self)
        self.release_bt.setText('暂停发布')

# 表格填充颜色
def fill_rpg(self, r, p, g, ):
    for k in range(5):
        if self.release_table.item(self.row_number_list[self.index],k):
            value = self.release_table.item(
                      self.row_number_list[self.index], k).text()
        else:
            value = ""
        newItem = QtWidgets.QTableWidgetItem(value)
        newItem.setBackground(QtGui.QColor(r, p, g))
        self.release_table.setItem(
    self.row_number_list[self.index],k,newItem)

# 将 CSV 文件写入数据表格
def importcsv_def(self):
    mkdir('temp')
    if os.path.exists('./temp/dispatch.csv'):
```

```
            # 清空现有数据
            self.release table.setRowCount(0)
            # 读取数据
            flie = open('./temp/dispatch.csv','r',encoding='gb18030')
            csv reader = csv.reader(flie)
            for index, row in enumerate(iter(csv reader)):
                if index != 0:
                    self.release table.setRowCount(
                                            self.release table.rowCount() + 1)
                    rownumber = self.release table.rowCount()
                    for i in range(5):
                        newItem = QtWidgets.QTableWidgetItem(row[i])
                        self.release table.setItem(rownumber-1,i,newItem)
            self.state value.setPlainText(
                         '导入成功' + '\n' +self.state_value.toPlainText())
            flie.close()
        else:
            self.state value.setPlainText(
              '找不到文件：dispatch.csv'+'\n' + self.state value.toPlainText())
        self.release table.resizeRowsToContents()

    # 将表格的数据导出到 CSV 文件
    def exportcsv def(self):
        mkdir('temp')
        temp list = []
        rownumber = self.release table.rowCount()
        f = open('temp/dispatch.csv','w',newline='',encoding='gb18030')
        writer = csv.writer(f)
        writer.writerow(['账号','密码','内容','图片','定时发布'])
        for i in range(rownumber):
            for k in range(5):
                value = ''
                ojb = self.release_table.item(i, k)
                if ojb:
                    # 先判断 ojb 是否为 None，在判断 ojb 的值是否为空
                    if ojb.text():
                        value = ojb.text()
                temp_list.append(value)
            writer.writerow(temp list)
            temp list = []
        f.close()
        self.state_value.setPlainText('导出成功' +'\n' +
                                 self.state value.toPlainText())

    # 表格自动添加行数
    def tableset(self):
        rownumber = self.release table.rowCount()
        if rownumber == 0:
            self.release table.setRowCount(rownumber + 1)
        if self.release table.item(rownumber - 1, 0):
            if self.release table.item(rownumber - 1, 0).text():
                self.release table.setRowCount(rownumber + 1)
        self.release table.resizeRowsToContents()
```

```
    # 设置定时发送
    def set_time(self):
        getrow = self.release_table.currentRow()
        getrow = 0 if getrow < 0 else getrow
        # 提取日期
        get_date = str(self.dateEdit.date()).split('(')[1].
                   split(')')[0].strip()
        # 设置日期格式
        get_time = '-'.join(get_date.split(',')).replace(' ','')
        # 为日期添加小时和分钟
        get_time += ' '+ self.hour.currentText()+
                    ':' + self.minute.currentText()
        # 计算设定的时间与现在的时间差
        now = datetime.datetime.now().strftime('%Y-%m-%d %H:%M')
        time_difference = datetime.datetime.strptime(
                          get_time, '%Y-%m-%d %H:%M')-
                          datetime.datetime.strptime(now,'%Y-%m-%d %H:%M')
        # 判断时间是否符合微博延时发送，若符合则写入对应的表格
        if time_difference.days >= 0:
            time_seconds = time_difference.seconds
            if time_seconds>=300 or time_difference.days>0:
                newItem = QtWidgets.QTableWidgetItem(get_time)
                self.release_table.setItem(getrow, 4, newItem)
                self.release_table.resizeRowsToContents()
            else:
                self.state_value.setPlainText('只能发 5 分钟后的定时微博哦。'+
                                    '\n' + self.state_value.toPlainText())
        else:
            self.state_value.setPlainText('只能发 5 分钟后的定时微博哦。'+
                                    '\n' + self.state_value.toPlainText())

    # 验证代理 IP
    def check_proxy(self):
        proxy_text = ''
        # 获取代理 IP 单号
        conf = configparser.ConfigParser()
        if os.path.exists('./temp/conf.ini'):
            conf.read('./temp/conf.ini')
            if 'config' in conf.keys():
                temp = conf['config']
                if 'proxies' in temp.keys():
                    proxy_text=conf['config']['proxies'].strip()
        if proxy_text:
            # 获取代理 IP
            url = 'http://api.ip.data5u.com/socks/get.html?
                  order='+proxy_text+'&json=1&type=1&sep=3'
            r = requests.get(url)
            info = r.json().get('data', '')
            if info:
                ip = info[0].get('ip')
                port = info[0].get('port')
                self.proxies=dict(http='http://'+str(ip)+':'+str(port))
            # 判断 IP 代理是否过期
            if not r.json().get('success', ''):
```

```
                self.proxies = {}
                self.warm_info = '单号未充值或者单号已经到期' + '\n'
            else:
                self.warm_info = '验证成功' + '\n'
        else:
            self.warm_info = '请设置你的代理 IP 单号' + '\n'
        self.state_value.setPlainText(self.warm_info +
                                      self.state_value.toPlainText())

    # 添加图片
    def showDialog(self):
        # 打开文件对话框
        foldername = QtWidgets.QFileDialog.getExistingDirectory(
                        self, '请选择图片所在文件夹', './')
        result = ''
        if foldername:
            pathDir = os.listdir(foldername)
            for allDir in pathDir:
                if len(self.pic_list) < 9 and
                   ('.jpg' in allDir or '.png' in allDir or '.gif' in allDir):
                    child = os.path.join('%s/%s' % (foldername, allDir))
                    result = result + child + ','
                    get_value = self.state_value.toPlainText()
                    get_value += child + ' 添加成功' + '\n'
                    self.state_value.setPlainText(get_value)
                    self.pic_list.append(child)
        # 在已选的数据行里写入图片路径
        getrow = self.release_table.currentRow()
        getrow = 0 if getrow < 0 else getrow
        newItem = QtWidgets.QTableWidgetItem(result)
        self.release_table.setItem(getrow, 3, newItem)
        self.release_table.resizeRowsToContents()
        self.pic_list = []

    # 返回主界面
    def show_win(self):
        self.setFixedSize(self.width(), self.height())
        self.setWindowFlag(QtCore.Qt.WindowMaximizeButtonHint,False)
        self.show()
    def close_win(self):
        self.close()

# 文件运行入口
if __name__ == '__main__':
    app = QtWidgets.QApplication(sys.argv)
    ex = weibo_release_logic()
    ex.show_win()
    sys.exit(app.exec_())
```

在 release.py 里，除了重写初始化函数__init__之外，还自定义了 13 个类方法，每个函数方法所实现的功能说明如下：

（1）初始化函数__init__()为软件的功能按钮做初始化处理，如设置数据表格自动增加行数、

各个按钮绑定功能函数以及初始化类属性等。

（2）keyPressEvent()定义键盘事件，当按下键盘的“Delete”键就会删除数据表格的当前选中行。参数 e 代表当前事件对象。

（3）release_weibo()实现微博登录和微博发送功能，实现过程说明如下：

- 调用 check_proxy 方法来检测代理 IP 的单号是否可用，若可用，则将 IP 地址写入类属性 proxies。
- 读取数据表格当前行的微博账号密码，并与类属性 session_dict 对比，判断当前微博账号是否已有登录对象 session。由于数据表格可以让同一微博账号发送多条微博，通过该判断能省去多次重复登录操作。
- 根据微博登录状态执行相应的处理。比如登录成功，状态码为 1000，则执行微博发布功能；如果状态码为 1001，说明微博账号密码错误或者验证码识别错误；如果状态码为 1002，说明验证码账号密码错误或者余额不足。
- 如果用户登录成功，则从登录对象 session 里获取微博用户信息，并从数据表格当前行的“定时发布”提取时间参数 addtime，这些都是用于构建微博发布的请求参数。
- 判断数据表格当前行的“图片”是否设有图片路径，若不为空，则将图片路径以列表表示，并且调用 upload_pic()方法向微博服务器上存图片，生成并获取相应的图片 ID，这也是用于构建微博发布的请求参数。
- 完成请求参数的构建，接着是调用 send_weibo()方法实现微博发送功能，该方法共有 8 个参数，这些参数的参数值分别来自软件界面的数据和用户登录信息。
- 微博发送成功后，数据表格当前行将填充成绿色并在状态文本框写入信息，若发送失败，则在状态文本框写入相应的信息。
- 当用户登录失败，状态文本框写入具体的错误信息，同时根据状态码在数据表格里填充相应的颜色。若状态码为 1001，数据表格填充成黄色；若状态码为 1002，数据表格填充成棕色。

（4）timerEvent()与微博采集的 timerEvent()相同，这是进度条对象特定的方法，该方法的代码结构与微博采集的 timerEvent()相同，此处不再详细讲述。

（5）timer_and_weibo()是“发布”按钮的功能函数，它的作用与微博采集的 collect_weibo_data()相同。该方法除了控制进度条对象 timer 的运行状态之外，还遍历数据表格的所有非空行数，对每一行的数据进行简单的清洗和判断，如果当前行的数据不符合判断条件，则填充上相应的颜色，表示当前微博不符合发送规则。

（6）fill_rpg()是表格的颜色填充方法，主要在 release_weibo()方法调用。实现过程是遍历当前行的每个单元格，再对每个单元格进行颜色填充。

（7）importcsv_def()是绑定“导入 CSV”按钮的功能函数，将数据表格的数据写入 dispatch.csv 文件，文件编码设为 gb18030，这样可以解决 Excel 读取 CVS 的乱码问题。

（8）exportcsv_def()是绑定“导出 CSV”按钮的功能函数，将 dispatch.csv 文件内容写入数据表格，文件编码设为 gb18030。

（9）tableset()是根据表格的 currentCellChanged 事件而决定是否新增行数，currentCellChanged 是当表格内容发生变化时而触发的事件。tableset()是判断表格总行数，如果行数为 0，则新增一行；

如果行数不为 0 并且最后一行的“账号”不为空，表格也会自动新增一行。

（10）set_time()是时间控件 QDateEdit 和 QCombobox 的功能函数，该方法是读取三个控件的时间并与当前时间进行计算，如果计算结果符合微博延时发送规则，将控件的时间写入表格的“定时发布”，否则提示错误信息。

（11）check_proxy()是读取配置文件的代理 IP 单号并向第三方平台发送请求，获取代理 IP 地址，从而验证代理 IP 服务是否正常使用。

（12）showDialog()是打开本机系统的文件对话框，让使用者选择图片文件夹，然后读取文件夹的图片路径并写入表格的“图片”。

（13）show_win()和 close_win()的作用与相关服务界面的作用一致，此处就不再重复讲述。

总的来说，微博发布的代码结构与微博采集的代码结构是相似的，两者都是使用进度条功能来执行相应的爬虫程序。而微博发布界面涉及了表格的编辑和颜色填充等操作，因此它比微博采集界面更为复杂，复杂程度在于软件的功能开发，并非爬虫程序。

运行 release.py 文件，在界面的数据表格分别新增 4 条数据，这 4 条数据的微博账号是相同的，并且设置了不同的发布方式。当单击“发布”按钮后，软件会对每条数据的内容进行判断并填充相应的颜色，如果颜色为绿色，说明微博发布成功，若为棕色，则代表数据内容不符合微博发送要求或者验证码账号密码错误或者余额不足，若为黄色，说明微博账号密码错误或者验证码识别错误，如图 21-17 所示。

发布内容					
	账号	密码	内容	图片	定时发布
1	17728115403		这个是正确的，但是定时的		2018-11-3 17:30
2	17728115403		这个是正确的，并且即时发布	D://timg.jpg,	
3	17728115403				2017/10/19 18:50
4	12312414	12123123	这个密码错误		
5					

图 21-17　微博发布功能

21.7　微博爬虫功能

在微博采集和微博发布的功能代码里分别调用 collect_weibo()、send_weibo()、upload_pic()和 login()函数。这些爬虫函数定义在 weibo.py 文件中，函数的实现过程在第 20 章已有详细讲述，在本节中，我们对爬虫函数进行细微的调整，使得爬虫函数能与软件相互结合使用。weibo.py 文件的代码如下：

```
import time
import base64
import rsa
import math
```

```
import random
import binascii
import requests
import re, json, urllib, csv, datetime, os
from weibo verify code import code verificate
from bs4 import BeautifulSoup
from concurrent.futures import ThreadPoolExecutor

#############
# 登录微博
#############
index_url = "http://weibo.com/login.php"
yundama username = 'beeto'
yundama password = 'beeto123'
verify code path = ''

def get pincode url(pcid):
    size = 0
    url = "http://login.sina.com.cn/cgi/pin.php"
    pincode_url = '{}?r={}&s={}&p={}'.format(url,
                  math.floor(random.random() * 100000000),
                  size, pcid)
    return pincode url

def get img(url, headers):
    resp = requests.get(url,headers=headers,stream=True)
    global verify code path
    mkdir('temp/code')
    verify code path='./temp/code/%s.png'%(str(int(time.time()*1000)))
    with open(verify code path, 'wb') as f:
        for chunk in resp.iter_content(1000):
            f.write(chunk)

def get su(username):
    """
    对 email 地址和手机号码先 javascript 中 encodeURIComponent
    对应 Python 3 中的是 urllib.parse.quote_plus 然后在 base64 加密后 decode
    """
    username quote = urllib.parse.quote plus(username)
    username base64=base64.b64encode(username quote.encode("utf-8"))
    return username base64.decode("utf-8")

# 预登陆获得 servertime, nonce, pubkey, rsakv
def get_server_data(su, session, headers, proxies):
    pre url = "http://login.sina.com.cn/sso/prelogin.php?
              entry=weibo&callback=sinaSSOController.
              preloginCallBack&su="
    pre url = pre url+su+"&rsakt=mod&checkpin=1&
             client=ssologin.js(v1.4.18)& ="
    prelogin url = pre url + str(int(time.time() * 1000))
    pre data res=session.get(prelogin url,headers=headers,proxies=proxies)
    sever data = eval(pre data res.content.decode("utf-8").
                replace("sinaSSOController.preloginCallBack",''))
    return sever data
```

```
    # 这一段用户加密密码，需要参考加密文件
    def get_password(password,servertime,nonce,pubkey):
        rsaPublickey = int(pubkey,16)
        # 创建公钥
        key = rsa.PublicKey(rsaPublickey, 65537)
        # 拼接明文js加密文件中得到
        message=str(servertime)+'\t'+str(nonce)+'\n'+str(password)
        message = message.encode("utf-8")
        # 加密
        passwd = rsa.encrypt(message, key)
        # 将加密信息转换为16进制
        passwd = binascii.b2a_hex(passwd)
        return passwd

    def login(username, password, proxies={}):
        # 构造 Request headers
        verify_code = 'Nocode'
        agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
                 Gecko/20100101 Firefox/41.0'
        headers = {
            'User-Agent': agent
        }
        # 新建会话
        session = requests.session()
        # su 是加密后的用户名
        su = get_su(username)
        sever_data = get_server_data(su,session,headers,proxies)
        servertime = sever_data["servertime"]
        nonce = sever_data['nonce']
        rsakv = sever_data["rsakv"]
        pubkey = sever_data["pubkey"]
        password_secret=get_password(password,servertime,nonce,pubkey)
        postdata = {
            'entry': 'weibo',
            'gateway': '1',
            'from': '',
            'savestate': '7',
            'useticket': '1',
            'pagerefer': "http://login.sina.com.cn/sso/logout.php?
                          entry=miniblog&r=http%3A%2F%2Fweibo.com
                         %2Flogout.php%3Fbackurl",
            'vsnf': '1',
            'su': su,
            'service': 'miniblog',
            'servertime': servertime,
            'nonce': nonce,
            'pwencode': 'rsa2',
            'rsakv': rsakv,
            'sp': password_secret,
            'sr': '1366*768',
            'encoding': 'UTF-8',
            'prelt': '115',
            'url': 'http://weibo.com/ajaxlogin.php?framelogin=1&callback=
                   parent.sinaSSOController.feedBackUrlCallBack',
```

```
            'returntype': 'META'
        }
        try:
            need_pin = sever_data['showpin']
            if need_pin == 1:
                if not yundama_username:
                    raise Exception('由于本次登录需要验证码，请配置顶部位置云打码的
                                    用户名{}及相关密码'.format(yundama_username))
                pcid = sever_data['pcid']
                postdata['pcid'] = pcid
                img_url = get_pincode_url(pcid)
                get_img(img_url, headers)
                verify_code = code_verificate(yundama_username,
                                  yundama_password, verify_code_path)
                postdata['door'] = verify_code
            login_url = 'http://login.sina.com.cn/sso/login.php?
                          client=ssologin.js(v1.4.18)'
            login_page = session.post(login_url,data=postdata,
                                      headers=headers,proxies=proxies)
            login_loop = (login_page.content.decode("GBK"))
            pa = r'location\.replace\([\'"](.*?)[\'"]\)'
            loop_url = re.findall(pa, login_loop)[0]
            login_index=session.get(loop_url,headers=headers,proxies=proxies)
            uuid = login_index.text
            uuid_pa = r'"uniqueid":"(.*?)"'
            uuid_res = re.findall(uuid_pa, uuid, re.S)[0]
            web_weibo_url = "http://weibo.com/%s/profile?
                              topnav=1&wvr=6&is_all=1" % uuid_res
            weibo_page=session.get(web_weibo_url,headers=headers,
                                   proxies=proxies)
            weibo_pa = r'<title>(.*?)</title>'
            user_name = re.findall(weibo_pa, weibo_page.content.
                         decode("utf-8", 'ignore'), re.S)[0]
            # 获取用户信息
            response = weibo_page.text
            person_info = {}
            if '$CONFIG' in response:
                person_info['nick'] = response.split(
                              "$CONFIG['nick']='")[1].split("';")[0]
                person_info['watermark'] = response.split(
                              "$CONFIG['watermark']='")[1].split("';")[0]
                person_info['location'] = response.split(
                              "$CONFIG['location']='")[1].split("';")[0]
                person_info['uid'] = response.split(
                              "$CONFIG['uid']='")[1].split("';")[0]
                person_info['domain'] = response.split(
                              "$CONFIG['domain']='")[1].split("';")[0]
                person_info['oid'] = response.split(
                              "$CONFIG['oid']='")[1].split("';")[0]
            print('登陆成功，你的用户名为：' + user_name)
            return {'session':session,'info':person_info,'code':'1000'}
        except:
            if verify_code != 'Nocode' and verify_code == '':
                # 打码平台账号密码错误或者余额不足
                return {'code':'1002'}
```

```
        else:
            # 微博账号密码错误或者验证码识别错误
            return {'code':'1001'}

##############
# 发送微博
##############
# 获取上传图片 id
def upload_pic(watermark,nick,session,file_list=[],proxies={}):
    return_result = []
    agent = 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_5)
               AppleWebKit/537.36 (KHTML, like Gecko)
               Chrome/60.0.3112.113 Safari/537.36'
    headers = {
        'User-Agent': agent
    }
    if len(file_list) > 0 and len(file_list) < 10:
        for i in file_list:
            url='http://picupload.service.weibo.com/interface/pic_upload.php?
                 mime=image/png&data=base64&url=weibo.com/' + str(watermark)+
                '&markpos=1&logo=&nick=@'+str(nick)+'&marks=1&app=miniblog'
            files = {
                'b64_data':base64.b64encode(open(i, "rb").read())
            }
            r=session.post(url,files=files,headers=headers,proxies=proxies)
            try:
                get_picid = json.loads(r.text.split('</script>')[1])
                               ['data']['pics']['pic_1']['pid']
                return_result.append(get_picid)
            except:
                pass
    return return_result

# 发送微博。send_type 判断是否发送图片
# pic_id_list 是上传图片后所生成的 Id
def send_weibo(watermark,location,value,session,addtime='',
                    pic_id_list=[],send_type='words',proxies={}):
    headers = {
        'Referer': 'https://weibo.com/' + str(watermark) + '/home',
        'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_5)
                        AppleWebKit/537.36 (KHTML, like Gecko)
                        Chrome/60.0.3112.113 Safari/537.36'}
    data = {}
    # 发送文字
    if send_type == 'words':
        data = {'location': location, 'text': value, 'appkey': '',
                 "style_type": "1", "pic_id": "", 'tid': '',
                 "pdetail": "", 'addtime': addtime,
                 "rank": "0", "rankid": "", 'module': 'stissue',
                 'pub_type': 'dialog', 'pub_source': 'main_', '_t': '0'
                }
    # 发送图片
    elif send_type != 'words' and pic_id_list:
        pic_id = ''
```

```
        for i in pic id list:
            pic id += i + '|' if len(pic id list) > 1 else i
        # 去除最后的|
        if pic_id[-1] == '|':
            pic id = pic id[0:len(pic id) - 1]
        data = {'location': location, 'text': value, 'appkey': '',
                'style type': '1', 'pic id': pic id, 'tid': '',
                'pdetail': '', 'gif ids': '', 'updata img num':
                str(len(pic id list)), 'addtime': addtime,
                'rank': "0", 'rankid': '', 'module': 'stissue',
                'pub type': 'dialog', 'pub source': 'main ',' t':'0'
              }
    url = 'https://www.weibo.com/aj/mblog/add?
           ajwvr=6&  rnd=%s' % (int(time.time()*1000))
    r=session.post(url,data=data,headers=headers,proxies=proxies)
    if r.status code==200:
        return True
    else:
        return False

#############
# 采集数据
#############
def mkdir(path):
    # 去除首位空格
    path = path.strip()
    # 去除尾部 \ 符号
    path = path.rstrip("\\")
    # 判断路径是否存在
    isExists = os.path.exists(path)
    # 判断结果
    if 'csv' in path and isExists==False:
        f=open('temp/data.csv','w',newline='',encoding='gb18030')
        writer = csv.writer(f)
        writer.writerow(['用户','文本内容','图片','视频','采集日期'])
        f.close()
    if not isExists and 'csv' not in path:
        os.makedirs(path)

# 线程爬取视频文件
def more thread(get video value, video path):
    if get video value:
        url = get video value['action-data'].
              split('video src=')[1].split('&cover img=')[0]
        url = 'http:' + urllib.parse.unquote(url)
        try:
            temp value = requests.get(url)
            video = open('temp/video/'+video path,'wb')
            video.write(temp_value.content)
            video.close()
        except:
            pass

# 线程爬取图片
```

```
def thread img(k, img path):
    img r = requests.get('http:' + k['src'])
    img = open('temp/image/' + img path, 'wb')
    img.write(img_r.content)
    img.close()

# 获取搜索内容
def collect weibo(keyword,session,pagenumber=1,proxies={},
                   get img=False, get video=False):
    now = datetime.datetime.now().strftime('%Y-%m-%d')
    keyword = urllib.parse.quote(keyword)
    url = 'https://s.weibo.com/weibo?q=' + keyword +
          '&Refer = SWeibo bo&page=%s' % (str(pagenumber))
    agent = 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
              Gecko/20100101 Firefox/41.0'
    headers = {
        'User-Agent': agent
    }
    r = session.get(url, headers=headers, proxies=proxies)
    get_value = r.text.replace('\/', '/')
    soup = BeautifulSoup(get value, 'html5lib')
    # 定位用户信息
    get_info = soup.find_all('div', class_="content")
    # 生成素材文件夹
    mkdir('temp')
    mkdir('temp/image')
    mkdir('temp/video')
    mkdir('temp/data.csv')
    for i in get info:
        # 获取文字全部内容
        get comment = i.find all('p',class ='txt')
        if get comment:
            if len(get_comment) > 1:
                get comment = get comment[-1]
            else:
                get comment = get comment[0]
            # 输出全部文字内容
            comment = get comment.getText().strip()
            # 获取用户信息
            get user = i.find('a',class ='name')
            if get_user:
                user name = get user.getText().strip()
            else:
                user name = ''
            # 爬取图片
            img_path_list = ''
            if get img:
                # 获取图片内容
                get_img_value = i.find('ul', class_='m3')
                # 输出图片
                if get img value:
                    get img value = get img value.find all('img')
                    for k in get img value:
                        img_path = str(int(time.time() * 1000)) + '.jpg'
```

```
            img_path_list = img_path_list + img_path + '/'
            pool = ThreadPoolExecutor(max_workers=1)
            pool.submit(thread_img, k, img_path)
# 爬取视频
video_path = ''
if get_video:
    # 输出视频
    get_video_value = i.find('a',class_='WB_video_h5')
    if get_video_value:
        pool = ThreadPoolExecutor(max_workers=1)
        video_path = str(int(time.time()*1000))+'.mp4'
        pool.submit(more_thread,get_video_value,video_path)
# 用于生成csv
f = open('temp/data.csv','a',newline='',encoding='gb18030')
writer = csv.writer(f)
writer.writerow([user_name,comment,img_path_list,video_path,now])
f.close()
```

上述代码定义了多个函数，但真正的爬虫函数只有 login()、send_weibo()和 collect_weibo()，其他函数都是被它们所调用。而爬虫函数可取第 20 章所实现的代码，并且在此基础上进行修改，具体说明如下：

（1）函数 login()新增函数参数 proxies，参数值默认为空字典，如果参数 proxies 为空，那么在登录的过程中会使用参数 proxies 作为代理 IP 进行账号登录。函数返回值新增状态码 code，在软件界面里根据 code 判断用户登录情况，方便使用者排查异常。

（2）函数 send_weibo()新增函数参数 send_type 和 proxies，参数值分别为 words 和空字典。参数 send_type 是判断微博发布是否带有图片发布；参数 proxies 作为代理 IP 进行微博发布。

（3）函数 collect_weibo()新增函数参数 proxies、get_img 和 get_video，参数值分别为空字典、False 和 False。参数 proxies 作为代理 IP 进行微博采集；get_img 和 get_video 是根据采集选项的勾选情况来决定是否爬取图片和视频。

此外，其他功能函数还需要结合爬虫函数的修改而进行相应的调整，最明显的调整是参数 proxies 的传递和使用。

21.8　本 章 小 结

本章主要讲述了爬虫软件的开发，这是爬虫开发的一个重要应用，最典型的爬虫软件有 12306 抢票软件、营销爬虫软件。营销软件主要有微博爬虫软件、淘宝爬虫软件和直播爬虫软件，这些软件可以实现多用户的批量操作，比如本书所讲述的微博用户批量发布微博以及直播的字幕批量刷屏等。

微博爬虫软件主要实现 4 个功能界面：软件主界面、相关服务界面、微博采集界面和微博发布界面。其中核心界面有微博采集界面和微博发布界面；相关服务界面为两个核心界面提供第三方服务；软件主界面是为整个软件提供运行入口，通过主界面实现界面之间的切换。

软件主界面共有三个按钮及一个微博的背景图，这三个按钮分别是发布、采集和相关服务，

当分别单击这三个按钮的时候，软件就会自动切换到相应的功能界面。

相关服务界面分为打码服务和代理服务，这些服务都是第三方网站提供，要使用这些服务，只需在界面上设置相关的账号信息即可。对于这些服务的使用，本书不做详细介绍，读者可单击软件的“购买打码服务”和“购买代理服务”按钮，进入官网了解使用方法。

微博采集界面是将热门微博的爬虫与软件开发相结合，这是爬虫软件的核心开发思想，使用者在软件设置的信息会以函数参数的形式传递给爬虫函数，爬虫根据用户设置的信息去执行相应的爬取操作。

微博发布界面与微博采集界面在设计上有一定的相似之处，但两者在功能上存在着明显的差异，主要体现在以下几点：

（1）数据表格具有编辑功能，如自动增加行数、整行删除、颜色填充等功能，而微博采集只具有数据查看功能。

（2）新增图片添加和定时功能，前者是打开系统的文件对话框；后者是由 QDateEdit 和 QCombobox 控件实现。

（3）不同账户的微博批量发布涉及到代理 IP 的验证与使用，微博定时发布的时间验证与设置。

第 22 章

Scrapy 爬虫开发

22.1 认识与安装 Scrapy

22.1.1 常见爬虫框架介绍

爬虫框架是为解决爬虫问题而设计的具有一定约束性的支撑结构。在此结构上，可以根据具体问题扩展、安插更多的组成部分，从而更迅速和方便地构建完整的解决问题的方案。

Python 常见的爬虫框架如下。

- Scrapy 框架：Scrapy 框架是一套比较成熟的 Python 爬虫框架，是使用 Python 开发的快速、高层次的信息爬取框架，可以高效地爬取 Web 页面并提取出结构化数据。
- PySpider 框架：PySpider 是以 Python 脚本驱动的抓取环模型爬虫框架。
- Crawley 框架：Crawley 也是 Python 开发的爬虫框架，该框架致力于改变人们从互联网中提取数据的方式。
- Portia 框架：Portia 是一款允许没有任何编程基础的用户可视化地爬取网页的爬虫框架。
- Newspaper 框架：Newspaper 是一款用来提取新闻、文章以及内容分析的 Python 爬虫框架。

爬虫框架能为项目开发起到规范作用，也因为如此，使其失去一定的灵活性。很多人会将 Requests 和 Scrapy 两者进行对比，前者是第三方库，后者是爬虫开发框架，尽管两者不在同一层次上，但还是有一定的对比性。

- 规范性：Scrapy 有自身的一套规则，自带功能模块能完成爬虫开发，各个功能代码划分明确。Requests 只规范数据爬取，不支持数据清洗和数据存储，需结合其他库一起使用才能完成爬虫开发。
- 灵活性：Scrapy 有较强的规范性，导致其灵活性比不上 Requests，对于一些设计不合理的网站或者较为特殊的网站，Requests 能针对其特殊性制定完善的解决方案。
- 适用范围：Scrapy 适用于大型爬虫开发项目，主要归功于其具有明确的规范性，便于开发

者对代码的维护和管理。Requests 对开发人员的编程习惯有较大影响，如果架构设计不合理或者更换开发人员，会使代码维护管理难以把控。

总的来说，无论是框架式开发还是非框架开发，都应针对项目的整体需求制定合理的开发设计方案。只要是合理的便是最好的，无论是框架与非框架，只是一个开发工具而已。

在 Python 中，开源爬虫框架很多，但并不需要掌握每一种爬虫框架，只需要深入掌握一种即可。大部分爬虫框架的实现方式都大同小异，基本上都是围绕爬虫开发流程（网页抓取、数据清洗、数据入库和异步并发处理等方面）设计而成的。

Scrapy 是一个为了爬取网站数据、提取结构性数据而编写的应用框架，主要应用在数据挖掘、信息处理或存储历史数据等一系列程序中。Scrapy 最初是为了页面抓取而设计的，也可以应用在获取 API 所返回的数据（例如 Amazon Associates Web Services）或者通用的网络爬虫中。

Scrapy 基于 Twisted 架构，使得可以级联多个操作，包括清理、组织、存储数据到数据库等。假设抓取一个网站，网站的每一页有上百数据，Scrapy 可以同时对这个网站发起 16 个或者更多请求，假如每个请求需要一秒钟来完成，相当于每秒钟爬取 16 个页面，每秒钟生成 1600 条数据，把这些数据同时存储入库，每条数据的存储需要 3 秒钟（假设时间），为了处理这 16 个请求，就需要运行 1600×3 = 4800 个并发的写入请求，对于一个传统的多线程程序来说，就需要转换成 4800 个线程，这会对系统造成极大的压力。对于 Scrapy 来说，只要硬件过关，4800 个并发请求是没有问题的。除此之外，Scrapy 还提供了 selectors（在 lxml 的基础上提供了更高级的接口），可以高效地处理不完整的 HTML 代码。

22.1.2 Scrapy 的运行机制

Scrapy 使用 Twisted 异步网络库来处理网络通信，架构清晰，并且包含各种中间件接口，可以灵活地完成各种需求。Scrapy 的整体架构如图 22-1 所示。

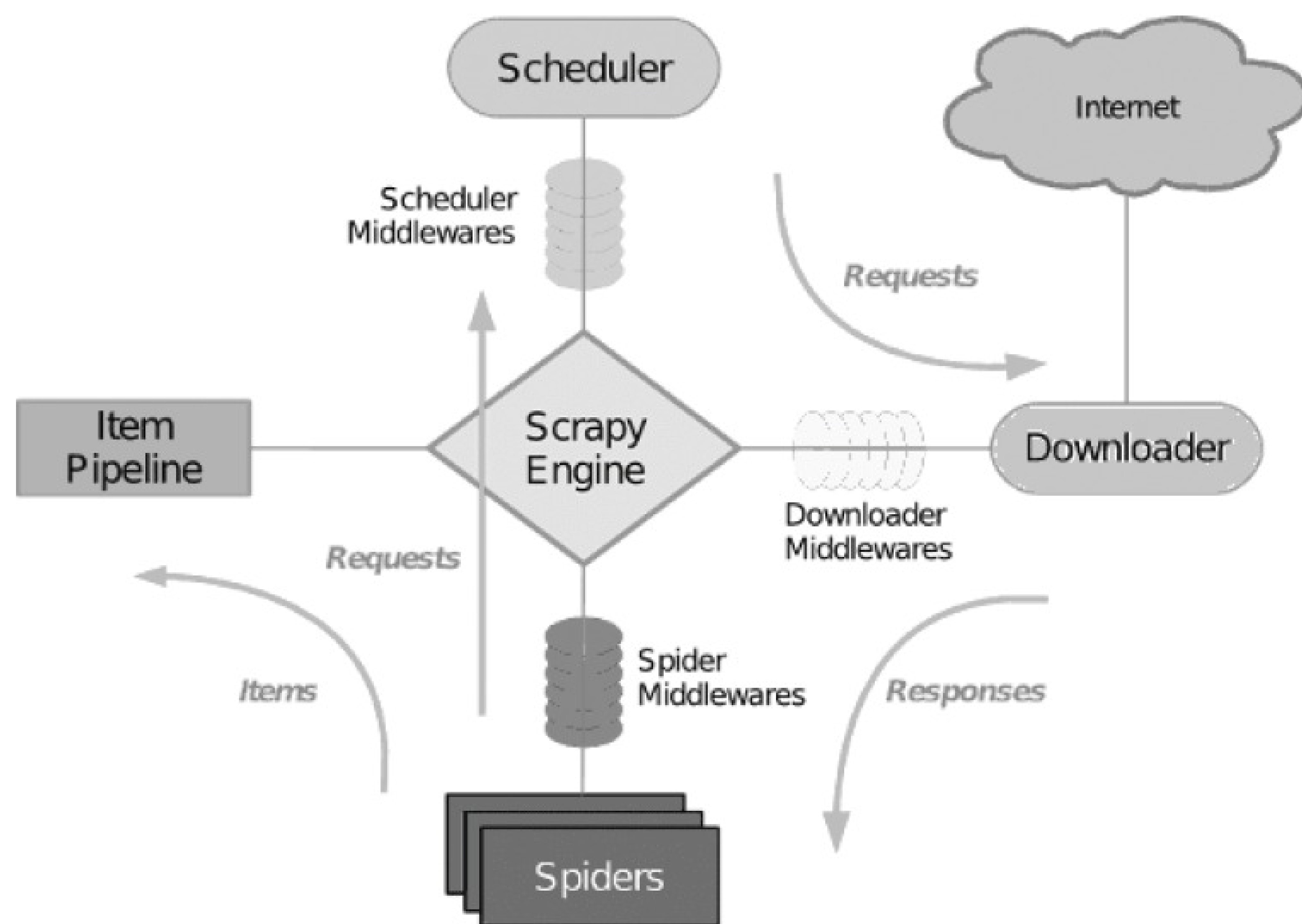


图 22-1 Scrapy 的运行机制

Scrapy 的运行机制大概如下：

（1）引擎从调度器中取出一个 URL（URL），用于接下来的抓取。

（2）引擎把 URL 封装成请求（Request）传给下载器，下载器把资源下载后封装成应答包（Response）。

（3）爬虫解析 Response。

（4）若解析出实体（Item），则交给实体管道进行进一步的处理。

（5）若解析出的是 URL，则把 URL 交给 Scheduler 等待抓取。

Scrapy 的运行离不开各个组件相互合作和调度。结合图 22-1，各个组件的功能说明如下。

- 引擎（Scrapy Engine）：处理整个系统的数据流，触发事务（框架核心）。
- 调度器（Scheduler）：接受引擎发过来的请求，压入队列中，并在引擎再次请求的时候返回。
- 下载器（Downloader）：用于下载网页内容，并将网页内容返回给蜘蛛（Scrapy 下载器的运行原理是基于 Twisted 框架实现的）。
- 爬虫（Spiders）：从特定的网页中提取自己需要的信息，即实体（Item）。也可以从中提取出 URL，让 Scrapy 继续抓取下一个页面。
- 项目管道（Item Pipeline）：负责处理爬虫从网页中抽取的实体，主要的功能是持久化实体、验证实体的有效性、清除不需要的信息。当页面被爬虫解析后，将被发送到项目管道，并经过几个特定的次序处理数据。
- 下载器中间件（Downloader Middlewares）：位于 Scrapy 引擎和下载器之间的框架，处理引擎与下载器之间的请求及响应。
- 爬虫中间件（Spider Middlewares）：介于 Scrapy 引擎和爬虫之间的框架，主要工作是处理蜘蛛的响应输入和请求输出。
- 调度中间件（Scheduler Middewares）：介于 Scrapy 引擎和调度器之间的中间件，用于处理从 Scrapy 引擎发送到调度器的请求和响应。

22.1.3　安装 Scrapy

在安装 Scrapy 之前，需要先安装 Twisted。Twisted 可以使用 pip 安装，但使用 pip 安装很容易出现错误，建议下载 Twisted 的 whl 文件安装（www.lfd.uci.edu/~gohlke/pythonlibs/），如图 22-2 所示。

Twisted, an event-driven networking engine.

- Twisted-18.9.0-cp27-cp27m-win32.whl
- Twisted-18.9.0-cp27-cp27m-win_amd64.whl
- Twisted-18.9.0-cp34-cp34m-win32.whl
- Twisted-18.9.0-cp34-cp34m-win_amd64.whl
- Twisted-18.9.0-cp35-cp35m-win32.whl
- Twisted-18.9.0-cp35-cp35m-win_amd64.whl
- Twisted-18.9.0-cp36-cp36m-win32.whl
- Twisted-18.9.0-cp36-cp36m-win_amd64.whl
- Twisted-18.9.0-cp37-cp37m-win32.whl
- Twisted-18.9.0-cp37-cp37m-win_amd64.whl

图 22-2　Twisted 版本信息

如 Twisted□18.9.0□cp37□cp37m□win_amd64.whl，cp37 是 Python 3.7 版本，amd64 代表 64 位系统。下载文件后保存在 E 盘，然后打开 CMD 窗口，将路径切换到 E 盘，输入安装指令：

```
E:\>pip install Twisted□18.9.0□cp37□cp37m□win_amd64.whl
```

完成 Twisted 安装后，可使用 pip 安装 Scrapy，安装指令如下：

```
pip install Scrapy
```

值得注意的是，最好先安装 Twisted，再安装 Scrapy。如果直接安装 Scrapy，在安装过程中就会出现报错信息：

```
building 'twisted.test.raiser' extension
error: Microsoft Visual C++ 14.0 is required. Get it with "Microsoft Visual
C++ Build Tools": http://landinghub.visualstudio.com/visual-cpp-build-tools
```

如果出现上述报错信息，用户先安装 Twisted，再重新安装 Scrapy 即可解决。完成 Scrapy 的安装后，打开 CMD 窗口并进入 Python 交互式命令行，输入以下代码检测是否安装成功：

```
>>> import scrapy
>>> scrapy.  version
'1.5.1'
```

22.2 Scrapy 爬虫开发示例

本节通过一个简单的项目讲解如何使用 Scrapy 实现爬虫开发，以百度知道的问题列表为例。在浏览器中打开网页（https://zhidao.baidu.com/list?cid=110）和开发者工具，查找并分析网页数据的生成方式。最终在 Doc 标签下找到数据所在位置，分析得知每条数据在标签<a>中，而标签<a>嵌套在标签<div>中，class 属性的值为 question-title，如图 22-3 所示。

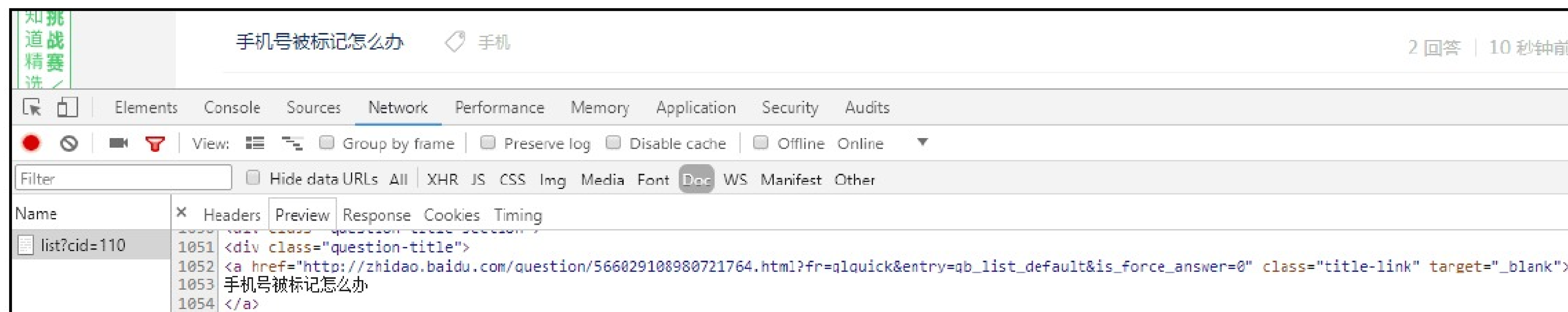


图 22-3　分析百度知道问题列表

根据简单分析，使用 Scrapy 完成上述开发需求。首先创建 Scrapy 项目，在 CDM（终端）下切换到 E 盘路径，本项目以“baidu”为项目名称，创建项目命令如下：

```
scrapy startproject baidu
```

创建项目后，可在 E 盘找到“baidu”文件夹，在 Pychram 下打开该文件夹，目录结构如图 22-4 所示。

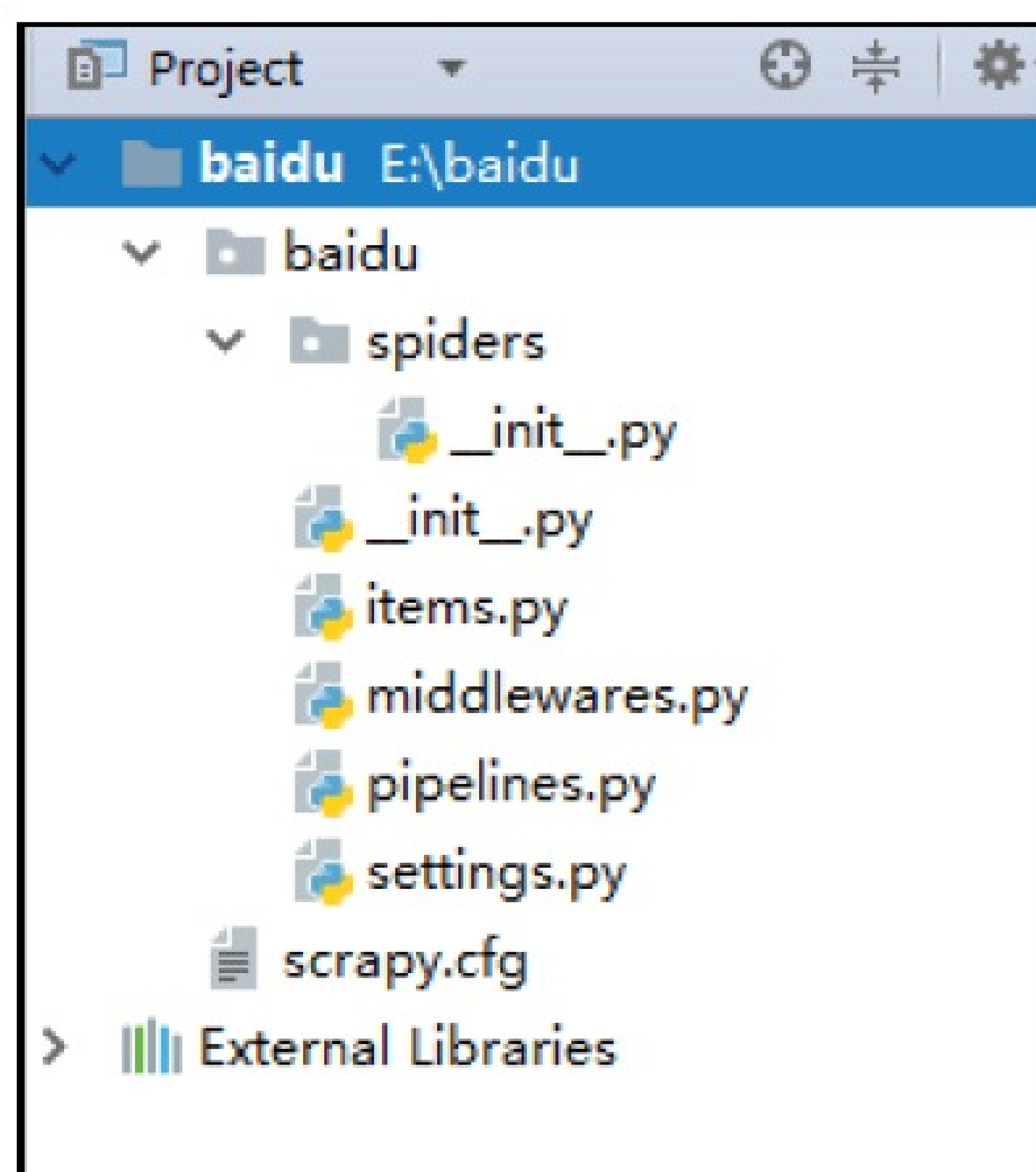


图 22-4　Scrapy 目录结构

项目文件说明如下。

- spiders（文件夹）：编写爬虫规则，实现数据爬取和数据清洗处理。
- items.py：数据定义和实例化，用于寄存清洗后的数据。
- middlewares.py：是介于 Scrapy 的 request/response 处理的钩子框架，用于全局修改 Scrapy request 和 response 的一个轻量、底层的系统。
- pipelines.py：执行保存数据的操作，数据对象来源于 items.py。
- setting.py：整个框架配置文件。
- scrapy.cfg：项目部署文件。

使用框架开发一般都有功能实现次序，但次序并不是固定不变的，很大程度上都根据开发人员的编程习惯来决定。Scrapy 的常用功能实现次序如下。

- setting.py：主要配置爬虫信息，如请求头、中间件和延时设置等。
- items.py：定义存储数据对象，主要衔接 spiders（文件夹）和 pipelines.py。
- pipelines.py：数据存储，数据格式以字典形式表现，字典的键是 items.py 定义的变量。
- spiders（文件夹）：编写爬虫规则。

下面按照上述实现次序讲解功能代码的编写。

步骤01 打开 setting.py，发现文件大部分内容已被注释，注释内容有配置代码、配置说明和相应的官方文档链接。本项目只需设置 Item Pipeline 和请求头即可，找到以下代码，将其注释去掉，其余代码不做任何操作：

```
# 指定数据入库的函数
ITEM_PIPELINES = {
   'baidu.pipelines.BaiduPipeline': 300,
}
# 设置请求头
DEFAULT_REQUEST_HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,application/xml;
            q=0.9, */*;q=0.8',
  'Accept-Language': 'en',
```

```
    }
```

配置信息说明如下：

- ITEM_PIPELINES 用于激活 pipelines.py 文件里的 BaiduPipeline 类，作用是告诉 Scrapy 在执行数据存储的时候使用哪个类对象实现存储。BaiduPipeline 是 Scrapy 项目自动生成的类，开发者也可根据实际需求添加或删除配置内容。
- DEFAULT_REQUEST_HEADERS 用于激活请求头，当 Scrapy 向网站发送请求的时候，如果该请求没有指明请求头内容，就默认使用该配置作为这个请求的请求头。

步骤02 打开 items.py，Scrapy 已生成相关的代码及文档说明，开发者只需在此基础上定义类属性即可。本项目定义类属性 TitleName，代表问题列表中每条问题的内容。scrapy.Field()是 Scrapy 的特有对象，其主要作用是处理并兼容不同的数据格式，开发者在定义类属性时无须考虑爬取数据的数据格式，Scrapy 会对数据格式做相应处理。实现代码如下：

```
import scrapy
class BaiduItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    TitleName = scrapy.Field()
    pass
```

步骤03 打开 pipelines.py，Scrapy 已生成类 BaiduPipeline 和相关说明，类 BaiduPipeline 就是 setting.py 配置 ITEM_PIPELINES 的内容。数据存储主要在类方法 process_item()中执行，本项目以将数据存储在文本文档为例进行介绍，代码如下：

```
class BaiduPipeline(object):
    def process item(self, item, spider):
        # 参数 item 是 items.py 的对象
        # 以下代码自行编写
        file = open('E:\\data.txt', 'a')
        for i in item['TitleName']:
            value = i.replace("\n", "")
            file.write(value + "\r\n")
        file.close()
        # 以上代码自行编写
        # return 主要输出 item 内容，若不需要，则可注释掉
        return item
```

步骤04 spiders（文件夹）用于编写爬虫规则，可以在已有的__init__.py 文件中编写具体的爬虫规则，但实际开发可能有多个爬虫规则，所以建议一个爬虫规则用一个文件表示，这样便于维护和管理。回到项目中，我们创建文件 Spider_spiders.py，代码如下：

```
# 导入 items.py 的 BaiduItem，存放爬取数据
from baidu.items import BaiduItem
# Scrapy 自带数据清洗模块
from scrapy.selector import Selector
# Scrapy 搜索引擎
from scrapy.spider import Spider
# 爬虫规则，一个爬虫以类为实现对象
class Baispider(Spider):
```

```
    # 属性 name 必须设置，而且是唯一命名的，用于运行爬虫
    name = "Baidu know"
    # 设置允许访问域名
    allowed domains = ["baidu.com"]
    # 设置 URL
    start urls = [
        "https://zhidao.baidu.com/list?cid=110",
        "https://zhidao.baidu.com/list?cid=110102"
    ]
    # 函数 parse 处理响应内容，函数名不能更改。
    def parse(self, response):
        # 将响应内容生成 Selector，用于数据清洗
        sel = Selector(response)
        items = []
        # 定义 BaiduItem 对象
        item = BaiduItem()
        title = sel.xpath('//div[@class="question-title"]
                  /a/text()'). extract()
        for i in title:
            items.append(i)
        item['TitleName'] = items
        return item
```

上述代码说明如下：

（1）爬虫规则以类为实现单位，并继承父类 Spider，Spider 是 Scrapy 的爬虫引擎之一。

（2）属性 name 不能为空，其是程序运行入口，如果有多个爬虫规则，那么每个规则的属性 name 不能重复，否则 Scrapy 无法识别执行哪一个爬虫规则。

（3）allowed_domains 是设置允许访问的域名，如果为空，就说明对域名不做访问限制。

（4）start_urls 用于设置爬取对象的 URL，程序运行时会对 start_urls 遍历处理。

（5）类方法 parse()用于处理网站的响应内容，如果爬虫引擎是 Spider，方法名就不能更改。

完成上述功能开发后，使用 CMD（终端）启动程序，将路径切换到 E:\baidu，运行命令如下：

```
E:\baidu>scrapy crawl Baidu_know
```

scrapy crawl 是启动 Scrapy 的命令符，Baidu_know 是 Spider_spiders.py 文件的 Baispider 类属性 name。程序运行结果如图 22-5 所示。

从运行结果看出，程序对 start_urls 的 URL 遍历访问，并返回 None 对象。因为对 pipelines.py 的 return item 做了注释处理，如果去掉注释，就会返回爬取的数据内容。程序运行结束后，Scrapy 会将运行信息返回，开发人员可根据信息调整 setting.py 的并发数和延时配置。

Baispider 类继承自父类 Spider，在 Scrapy 中，大多数爬虫使用 Spider 类足以胜任，如果要爬取全站数据而且具有一定规则的网站，Spider 虽然可以实现，但实现过程相当复杂，这时我们需要更强大的武器 CrawlSpider。

```
n 127.0.0.1:6023
2017-12-11 17:48:17 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://zhida
o.baidu.com/robots.txt> (referer: None)
2017-12-11 17:48:17 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://zhida
o.baidu.com/list?cid=110> (referer: None)
2017-12-11 17:48:17 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://zhida
o.baidu.com/list?cid=110102> (referer: None)
2017-12-11 17:48:17 [scrapy.core.scraper] DEBUG: Scraped from <200 https://zhida
o.baidu.com/list?cid=110>
None
2017-12-11 17:48:17 [scrapy.core.scraper] DEBUG: Scraped from <200 https://zhida
o.baidu.com/list?cid=110102>
None
2017-12-11 17:48:17 [scrapy.core.engine] INFO: Closing spider (finished)
2017-12-11 17:48:17 [scrapy.statscollectors] INFO: Dumping Scrapy stats:
{'downloader/request_bytes': 789,
 'downloader/request_count': 3,
 'downloader/request_method_count/GET': 3,
 'downloader/response_bytes': 47129,
 'downloader/response_count': 3,
 'downloader/response_status_count/200': 3,
 'finish_reason': 'finished',
 'finish_time': datetime.datetime(2017, 12, 11, 9, 48, 17, 782106),
 'item_scraped_count': 2,
 'log_count/DEBUG': 6,
 'log_count/INFO': 7,
 'response_received_count': 3,
 'scheduler/dequeued': 2,
 'scheduler/dequeued/memory': 2,
 'scheduler/enqueued': 2,
 'scheduler/enqueued/memory': 2,
 'start_time': datetime.datetime(2017, 12, 11, 9, 48, 16, 876446)}
2017-12-11 17:48:17 [scrapy.core.engine] INFO: Spider closed (finished)
```

图 22-5　Scrapy 运行结果

CrawlSpider 也继承自父类 Spider，拥有父类 Spider 的全部属性，并有自身的独特属性。

（1）rules 是 Rule 对象的集合，用于匹配目标网站并排除干扰。

（2）parse_start_url 用于爬取起始响应，必须要返回 Item，Request 是其中之一。

以上述项目为例，使用 CrawlSpider 实现上述爬虫规则：首先在 spiders（文件夹）下创建文件 CrawlSpider_spiders.py，代码如下：

```
# 导入 items.py 的 BaiduItem，存放爬取数据
from baidu.items import BaiduItem
# Scrapy 自带数据清洗模块
from scrapy.selector import Selector
# 导入 CrawlSpider
from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors import LinkExtractor
# 爬虫规则，一个爬虫以类为实现对象
class Baispider(CrawlSpider):
# 属性 name 必须设置，而且是唯一命名的，用于运行爬虫
    name = "Baidu"
# 设置允许访问域名
    allowed domains = ["baidu.com"]
# 设置 URL
    start urls = [
        "https://zhidao.baidu.com/list?cid=110"
        ]
# 编写爬取规则
    rules = (
        Rule(LinkExtractor(allow=('zhidao.baidu.com/question/', ),
                               deny=(),), callback='parse item'),)
# 编写处理函数
    def parse_item(self, response):
```

```
        sel = Selector(response)
        items = []
        item = BaiduItem()
        title = sel.xpath('//span[@class="ask-title "]/text()').extract()
        for i in title:
            items.append(i)
        item['TitleName'] = items
        return item
```

上述代码与 Spider_spiders.py 的实现功能是一致的，但在逻辑处理上完全不同：

（1）Spider_spiders.py 继承自 Spider，运行方式是遍历 start_urls 的 URL，从每个 URL 获取数据，数据主要来源于 start_urls 的 URL。

（2）CrawlSpider_spiders.py 继承自 CrawlSpider，start_urls 的 URL 被访问后，获取其响应内容里的 URL 列表，再根据 rules 规则对得到的 URL 列表进行筛选，选出所有符合规则的 URL，并对符合规则的 URL 调用 callback 所指定的函数进行访问和处理。

CrawlSpider 类的 Rule 参数说明如下。

- allow：满足括号中的值会被提取，如果为空，就全部匹配，支持正则表达式实现模糊匹配。
- deny：与匹配值不匹配的 URL 不提取。
- allowed_domains：会被提取的 URL 的 domains。
- deny_domains：一定不会被提取 URL 的 domains。
- callback：指定回调函数处理符合筛选规则的 URL 的响应内容。

从运行逻辑分析，CrawlSpider 爬虫更适合全站数据爬取和通用爬虫开发。因为 rules 是 Rule 对象的集合，如果需要编写多个规则，就可以设置多个 Rule 对象，callback 所指定的函数也可以自行命名。相对 Spider 来说，CrawlSpider 在使用上较为灵活一些。

提　示

Spiders 的说明

Spider 是定义如何抓取某个网站（或一组网站）的类，包括如何执行抓取（访问 URL）以及如何从页面中提取结构化数据（抓取数据）。换句话说，Spider 是开发者自定义的类，用于为特定网站（在某些情况下是一组网站）抓取和解析页面。

Spider 的执行周期如下：

（1）抓取第一个 URL 的初始请求，然后指定一个回调函数，从请求的响应来调用回调函数，请求链接通过调用 start_requests()方法（该方法在默认情况下是 GET 方式），parse 方法作为回调函数处理请求链接返回的请求结果。

（2）在回调函数中，主要是解析响应（网页）内容，并将解析后的数据存储在 Item 对象中。如果解析的内容中需要产生多次请求，就可将 URL 传递给 Request 对象并指定某个回调函数，然后由 Scrapy 访问下载，通过指定的回调处理它们的响应。

（3）在回调函数中，通常使用选择器（也可以使用 BeautifulSoup、lxml 等第三方库）解析页面内容，并将解析的数据存储在 Item 对象中。

最后，从 Spider 返回的 Item 对象在 item pipeline 对象中进行数据存储。

提示（续）

Spider 的种类如下。

- scrapy.spiders.Spider：最简单的 Spider 类，其他的 Spider 也继承自该类（包括 Scrapy 其他定义的 Spider 以及开发者自定义的 Spider）。它不提供任何特殊的功能，只提供一个默认的 start_requests()方法，请求从 start_urls 开始，Spider 发送请求，并使用函数 parse 处理每个响应内容。
- scrapy.spiders.CrawlSpider：这是抓取常规网站最常用的 Spider，因其提供了一个方便的机制，可通过定义一组规则来跟踪 URL，适合全站数据爬取和通用爬虫开发。除了拥有 scrapy.spiders.Spider 全部属性之外，还有特定属性 rules 和 parse_start_url 方法。
- scrapy.spiders.XMLFeedSpider：用来爬取 XML 形式的网页内容，通过某个指定的节点来遍历。可使用 iternodes、xml 和 html 三种形式的迭代器，不过当内容比较多的时候，推荐使用 iternodes，可以节省内存、提升性能，不需要将整个 DOM 加载到内存中再解析，而使用 html 可以处理 XML 有格式错误的内容。
- scrapy.spiders.CSVFeedSpider：与 XMLFeedSpider 非常相似，其遍历 CSV 行数，在每个迭代中被调用的方法是 parse_row()。
- scrapy.spiders.SitemapSpider：SitemapSpider 通过使用 Sitemaps 发现网址并抓取网站，支持嵌套 Sitemap 和从 robots.txt 中发现 Sitemap 网址。这类爬虫主要用于搜索引擎开发，主要用于爬取整个网站的地图和全部 URL。

一般来说，目前大多数网站主要以 HTML 为主，Spiders 开发是以 Spider 和 CrawlSpider 为主，本书主要以这两者为讲述对象。

22.3 Spider 的编写

从 22.2 节的内容得知，Spider_spiders.py 爬虫规则的请求方式都是 GET 请求，但在实际开发中，我们还需要使用 POST 请求，那么如何在 Spider 中实现 POST 请求呢？下面我们来看看具体的实现方法。

首先创建一个新的项目，命名为 mySpider：

```
scrapy startproject mySpider
```

在项目里的 spiders（文件夹）中创建文件 post_spiders.py，代码如下：

```
from scrapy.selector import Selector
from scrapy.spider import Spider
import scrapy
class Baispider(Spider):
name = "Post spider"
allowed domains = []
start urls = [
"http://127.0.0.1:5000/",
]
# 爬虫入口——重写 start_requests 方法
```

```
# scrapy.FormRequest 是 POST 方式，formdata 是 POST 参数，callback 回调函数
def start requests(self):
return [scrapy.FormRequest(
self.start_urls[0],
formdata={"Python": "爬虫开发"},
callback=self.mypsot)]
# 回调函数
def mypsot(self, response):
data = Selector(response).xpath('//p/text()').extract()[0]
print(data)
```

整个项目 mySpider 只添加上述代码和文件，其余文件不做修改和添加。为了方便测试代码，我们在本地使用 Flask 搭建一个测试系统（Flask 安装：pip install flask），系统代码如下：

```
from flask import Flask, request
app = Flask(  name  )
# app.route 设置 URL 路径，methods 是请求方式
# hello_world 视图函数
@app.route('/', methods=['POST','GET'])
def hello world():
    # 判断请求方式，返回不同结果
    # POST 请求
    if request.method == 'POST':
        return "This is Post,your post data is " + request.form['Python']
    # GET 请求
    else:
        return 'Hello World!'
# 系统启动运行
if   name   == '  main  ':
    app.run()
```

将系统代码保存在 system.py 文件中，然后运行文件即可启动系统。系统启动后，运行 mySpider 项目，运行结果如图 22-6 所示。

```
n 127.0.0.1:6023
2017-12-12 18:18:58 [scrapy.core.engine] DEBUG: Crawled (404) <GET http://127.0.
0.1:5000/robots.txt> (referer: None)
2017-12-12 18:18:58 [scrapy.downloadermiddlewares.retry] DEBUG: Retrying <POST h
ttp://127.0.0.1:5000/> (failed 1 times): [<twisted.python.failure.Failure twiste
d.internet.error.ConnectionDone: Connection was closed cleanly.>]
2017-12-12 18:18:58 [scrapy.core.engine] DEBUG: Crawled (200) <POST http://127.0
.0.1:5000/> (referer: None)
This is Post,your post data is 爬虫开发
2017-12-12 18:18:58 [scrapy.core.engine] INFO: Closing spider (finished)
2017-12-12 18:18:58 [scrapy.statscollectors] INFO: Dumping Scrapy stats:
{'downloader/exception_count': 1,
 'downloader/exception_type_count/twisted.web._newclient.ResponseNeverReceived':
 1,
 'downloader/request_bytes': 817,
 'downloader/request_count': 3,
 'downloader/request_method_count/GET': 1,
 'downloader/request_method_count/POST': 2,
 'downloader/response_bytes': 532,
 'downloader/response_count': 2,
 'downloader/response_status_count/200': 1,
 'downloader/response_status_count/404': 1,
 'finish_reason': 'finished',
 'finish_time': datetime.datetime(2017, 12, 12, 10, 18, 58, 125681),
 'log_count/DEBUG': 4,
```

图 22-6 post_spiders 运行结果

可以看到，运行结果输出了“This is Post,your post data is 爬虫开发”，返回结果和 Flask 系统代码是一致的，说明在 Scrapy 中可重写 start_requests 来改写初始请求方式。

一个完整的爬虫会将 POST 和 GET 请求相互交错使用，而且每个请求都可能需要特定的请求头和 Cookies。以 mySpider 项目为例实现上述功能需求：在 mySpider 项目的 spiders（文件夹）下新建文件 get_post_spiders.py，代码如下：

```
from scrapy.selector import Selector
from scrapy.spider import Spider
import scrapy

class Baispider(Spider):
    name = "Get Post spider"
    allowed_domains = []
    start urls = [
        "http://127.0.0.1:5000/",
    ]
    # 定义请求头和 Cookies，两者皆以字典形式表示
    headers = {'User-Agent':'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
                             Gecko/20100101 Firefox/41.0',}
    cookies = {}

    # 处理第一次 GET 请求的响应内容，return 用于发送第二次 POST 请求
    def parse(self, response):
        data = Selector(response).xpath('//p/text()').extract()[0]
        print(data)
        return [scrapy.FormRequest(
            self.start urls[0],
            cookies=self.cookies,
            headers=self.headers,
            formdata={"Python": "爬虫开发"},
            callback=self.mypsot)]

    # 处理第二次 POST 请求的响应内容，return 用于发送第三次 GET 请求
    def mypsot(self, response):
        data = Selector(response).xpath('//p/text()').extract()[0]
        print(data)
        return scrapy.Request(self.start urls[0], cookies=self.cookies,
                      headers=self.headers, callback=self.myget)

    # 处理第三次 GET 请求的响应内容
    def myget(self, response):
        data = Selector(response).xpath('//p/text()').extract()[0]
        print(data)
```

从上述代码分析三次请求：

第一次 GET 请求是 Scrapy 默认 start_requests 实现的，回调函数是 parse。

第二次 POST 请求是在函数 parse 处理完第一次请求的响应内容后，通过 return 发送第二次请求，并设置请求头和 Cookies，回调函数是 mypsot。

第三次 GET 请求是在函数 mypsot 处理完第二次请求的响应内容后，通过 return 发送第三次请求，并设置请求头和 Cookies，回调函数是 myget。

打开 CMD 窗口，运行 get_post_spiders.py 的爬虫规则，运行结果如图 22-7 所示。

```
2017-12-13 09:47:06 [scrapy.core.engine] INFO: Spider opened
2017-12-13 09:47:06 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pag
es/min), scraped 0 items (at 0 items/min)
2017-12-13 09:47:06 [scrapy.extensions.telnet] DEBUG: Telnet console listening o
n 127.0.0.1:6023
2017-12-13 09:47:06 [scrapy.core.engine] DEBUG: Crawled (404) <GET http://127.0.
0.1:5000/robots.txt> (referer: None)
2017-12-13 09:47:06 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://127.0.
0.1:5000/> (referer: None)
Hello World!
2017-12-13 09:47:06 [scrapy.core.engine] DEBUG: Crawled (200) <POST http://127.0
.0.1:5000/> (referer: http://127.0.0.1:5000/)
This is Post,your post data is 爬虫开发
2017-12-13 09:47:07 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://127.0.
0.1:5000/> (referer: http://127.0.0.1:5000/)
Hello World!
2017-12-13 09:47:07 [scrapy.core.engine] INFO: Closing spider (finished)
2017-12-13 09:47:07 [scrapy.statscollectors] INFO: Dumping Scrapy stats:
```

图 22-7　get_post_spiders 运行结果

此外，Spiders 中的 CrawlSpider、XMLFeedSpider、CSVFeedSpider 和 SitemapSpider 都继承于父类 Spider，因此前面实现的功能适用于 Spiders 的所有类。

22.4　Items 的编写

数据抓取的主要目标是从非结构化来源（通常是网页）中提取结构化数据。Scrapy 可以将提取的数据作为 Python 字典返回，但 Python 字典缺乏结构，字典的键会在输入时出现拼写错误或者返回数据不一致，因此，Scrapy 提供了 Items 对象，用于管理和规范爬取数据，使其结构规范化。

Items 主要存放在项目文件 items.py 中，每个 Items 对象以类的形式声明和命名，类属性为 Items 的字段，也就是需要存储数据的元数据键（metadata key）。Items 可脱离 Scrapy 项目单独使用，为了更好演练，在 E 盘下创建文件 items.py，代码如下：

```
import scrapy
# Product 类继承自 Item 类
class Product(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field()
    stock = scrapy.Field()
    # last updated 指明了该字段的序列化函数
    last_updated = scrapy.Field(serializer=str)

if __name__ =="__main__":
product = Product(name='Desktop PC', price=1000)
print (product)
```

使用代码定义 Product 类，类属性 name、price、stock 和 last_updated 分别代表产品名称、价格、库存数和更新时间。Scrapy 声明字段无须考虑其数据类型，统一以 scrapy.Field()命名即可。运行 items.py 文件，输出结果如下：

```
{'name': 'Desktop PC', 'price': 1000}
```

除此之外，还可以对其进行读取和判断等操作。代码如下：

```
# 数据存储一
product = Product(name='Desktop PC',price=1000)
print (product)
# 数据存储二
item = Product()
item['name'] = 'Mac'
item['price'] = 2000
print(item)
# 读取数据内容一，若不存在，则会输出 None
print(item.get('name', 'None'))
print(item.get('stock', 'None'))
# 读取数据内容二，使用该方法读取，若不存在，则会提示 keyerror
print(item['name'])
# print(item['stock'])
# 判断是否存在字段，输出 True 或 False
print('name' in item)
print('stock' in item)
# 获取键值对
print(item.keys())
print(item.items())
```

22.5 Item Pipeline 的编写

当 Spiders 爬取的数据存放到 Items 之后，回调函数的 return（yield）返回 Items 对象，这时会触发 Item Pipeline 对 Items 对象的操作。Item Pipeline 主要存放在项目文件 pipelines.py 中，用于实现数据存储。

22.5.1 用 MongoDB 实现数据入库

以 22.2 节的项目为例，我们将数据存储介质由文本文档改为 MongoDB，MongoDB 的数据库结构信息如图 22-8 所示。

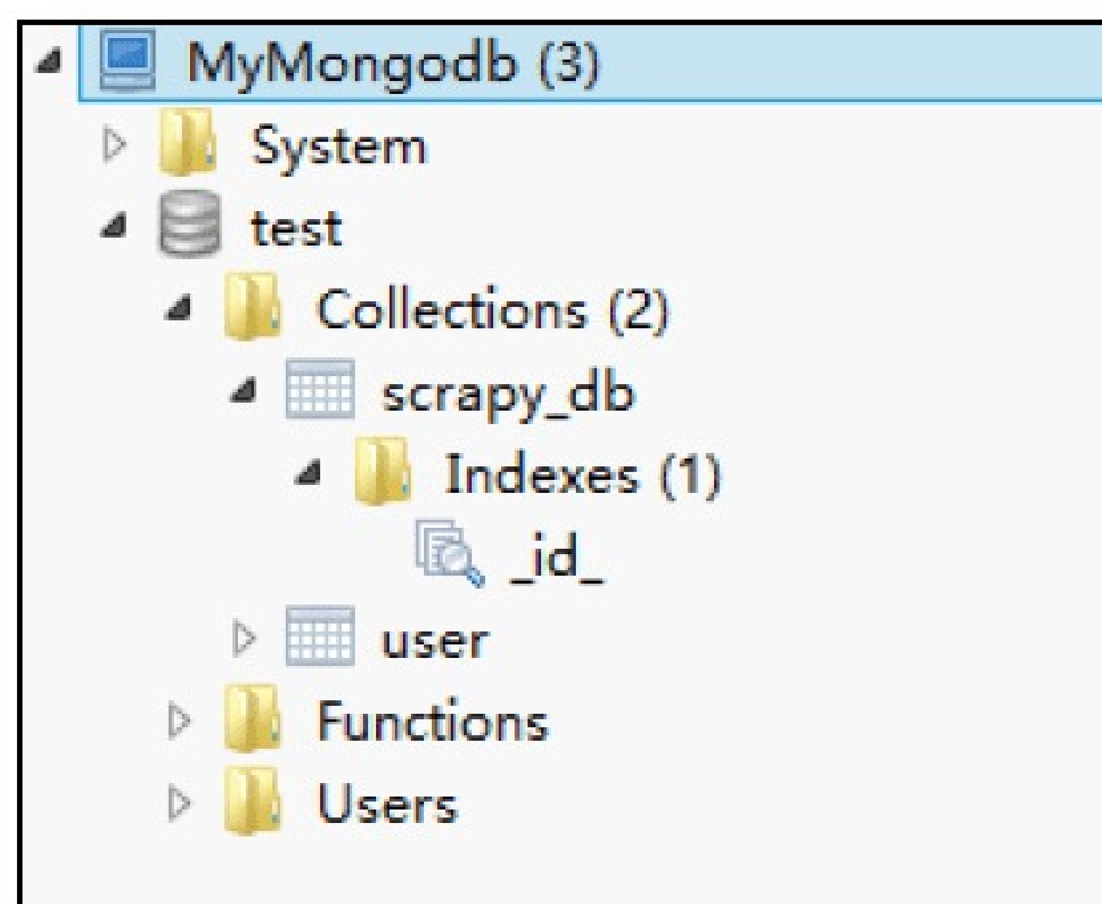


图 22-8 MongoDB 数据库

数据库信息如下。

（1）数据库所在服务器的 IP 地址：localhost。

（2）数据库端口：27017。

（3）数据库名：test

（4）Collection 名称：scrapy_db。

将数据库信息写入配置文件 setting.py，在 setting.py 中添加以下代码：

```
ITEM PIPELINES = {
   'baidu.pipelines.BaiduPipeline': 300,
}
# 数据库 IP
MONGODB SERVER = "localhost"
# 端口
MONGODB PORT = 27017
# Database 名称
MONGODB DB = "test"
# Collection 名称
MONGODB_COLLECTION = "scrapy_db"
```

完成了数据库信息的配置，接着编写 Item Pipeline 功能代码，pipelines.py 的代码如下：

```
# 导入 pymongo
from pymongo import MongoClient
# 导入 setting 配置信息
from scrapy.conf import settings

class BaiduPipeline(object):
    def   init   (self):
   # 连接数据库
        connection = MongoClient(
            settings['MONGODB SERVER'],
            settings['MONGODB PORT']
        )
        db = connection[settings['MONGODB DB']]
        self.collection = db[settings['MONGODB COLLECTION']]

    def process_item(self, item, spider):
   # 入库处理
        self.collection.insert(dict(item))
        return item
```

以 22.2 节的 Spider_spiders.py 爬虫规则运行程序，并查看数据库的数据信息，如图 22-9 所示。

从入库结果看到，生成了两条文档，每条文档对应 Spider_spiders.py 中 start_urls 的数量；每条 URL 有 30 条数据，也符合字段 TitleName 的数据量。从代码分析，在类 BaiduPipeline 的初始（__init__）函数中实现数据库连接功能，在函数 process_item 中实现数据入库。

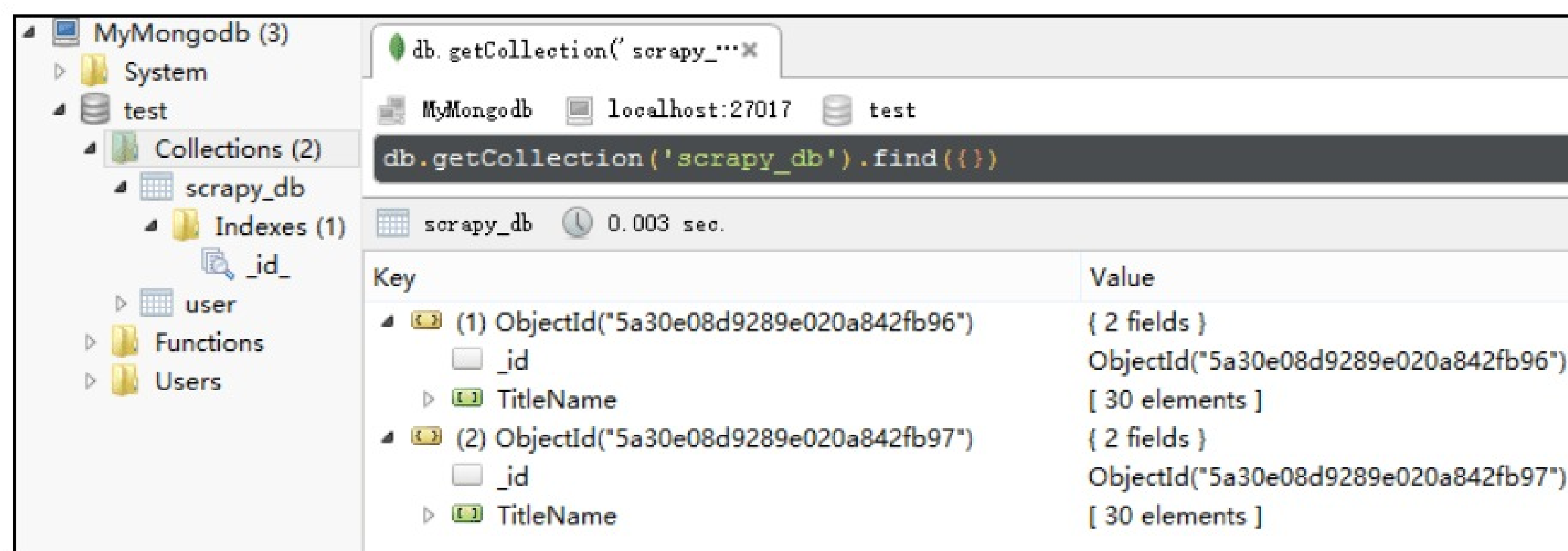


图 22-9　MongoDB 入库结果

22.5.2　用 SQLAlchemy 实现数据入库

上一节介绍了 MongoDB 入库，若要使用 SQLAlchemy 实现数据入库，实现方式大致相同。同样以 22.2 节的项目为例，以 MySQL 数据库为存储对象，MySQL 数据库信息如下：

（1）数据库所在服务器的 IP 地址：localhost。

（2）数据库用户：root。

（3）数据库密码：1234。

（4）数据库名：test。

将数据库信息写入配置文件 setting.py，在 setting.py 中添加代码如下：

```
# SQLAlchemy 连接数据库
MYSQL CONNECTION = 'mysql+pymysql://root:1234@localhost/test?charset=utf8'
编写 Item Pipeline 功能代码，pipelines.py 的代码如下：
# 导入 SQLAlchemy
from sqlalchemy import *
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative base
# 导入 setting 配置信息
from scrapy.conf import settings

# 定义映射类
Base = declarative base()
class scrapy db(Base):
    __tablename__ = 'scrapy_db'
    id = Column(Integer(), primary key=True)
    TitleName = Column(String(200))

class BaiduPipeline(object):
    def   init   (self):
        # 初始化，连接数据库
        conntion = settings['MYSQL CONNECTION']
        engine = create_engine(conntion, echo=False,pool_size=2000)
        DBSession = sessionmaker(bind=engine)
        self.SQLsession = DBSession()
        # 创建数据表
        Base.metadata.create_all(engine)
```

```
    def process_item(self, item, spider):
        # 入库处理
        self.SQLsession.execute(scrapy_db.__table__.insert(),
                 [{'TitleName': i} for i in item['TitleName']])
        self.SQLsession.commit()
        return item
```

以 22.2 节的 Spider_spiders.py 爬虫规则运行程序，并查看数据库的数据信息，如图 22-10 所示。



图 22-10　SQLAlchemy 入库结果

从图 22-10 得知，入库数据量和使用 MongoDB 入库的数据量是一致的。无论使用关系式数据库还是非关系式数据库，数据入库逻辑都相同。

根据数据入库逻辑总结 Item Pipeline 编写规则如下：

（1）使用 setting.py 配置数据库信息。数据库信息最好在 setting.py 中配置，这符合统一规范化开发要求。

（2）对 pipelines.py 的类初始化（__init__）函数实现数据库连接。如果使用 SQLAlchemy 入库，那么还需创建映射类映射数据表。

（3）由函数 process_item 实现数据入库。

22.6　Selectors 的编写

当抓取网页时，最常见的任务是从 HTML 源码中提取数据。Scrapy 提取数据有一套机制，被称作选择器（Seletors），通过特定的 XPath 或者 CSS 表达式来选择 HTML 中的某部分数据。当然，lxml 和 BeautifulSoup 也可以在 Scrapy 中担任数据清洗的角色。

Scrapy 选择器主要用于爬虫规则的编写。以 22.2 节的 Spider_spiders.py 爬虫规则为例进行介绍：

```
# 导入 items.py 的 BaiduItem，存放爬取数据
from baidu.items import BaiduItem
# Scrapy 自带数据清洗模块
from scrapy.selector import Selector
# Scrapy 搜索引擎
from scrapy.spider import Spider
```

```
# 爬虫规则，一个爬虫以类为单位

class Baispider(Spider):
    # 属性 name 必须设置，而且是唯一命名的，用于运行爬虫
    name = "Baidu know"
    # 设置允许访问域名
    allowed domains = ["baidu.com"]
    # 设置 URL
    start urls = [
        "https://zhidao.baidu.com/list?cid=110",
        "https://zhidao.baidu.com/list?cid=110102"
    ]
    # 函数 parse 处理响应内容，函数名不能更改
    def parse(self, response):
        # 将响应内容生成 Selector，用于数据清洗
        sel = Selector(response)
        items = []
        # 定义 BaiduItem 对象
        item = BaiduItem()
        title = sel.xpath('//div[@class="question-title"]
                  /a/text()'). extract()
        for i in title:
            items.append(i)
        item['TitleName'] = items
        return item
```

从上述代码可知，选择器的使用步骤如下。

（1）from scrapy.selector import Selector：导入 Selector 对象。

（2）sel = Selector(response)：声明 Selector 对象，并将响应内容加载该对象中。

（3）sel.xpath(XPath 语法).extract()：使用 XPath 对数据进行清洗，方法 extract()将数据以列表形式返回。

XPath 是一门用来在 XML 文件中选择节点的语言，也可以用在 HTML 中。CSS 是一门将 HTML 文档样式化的语言，选择器由它定义，并与特定 HTML 元素的样式相关。在两者的使用上，大部分开发人员偏向于 XPath。选择器主要掌握 XPath 或者 CSS 语法编写规则，本书以 XPath 语法为讲述重点。

XPath 使用路径表达式来选取 XML 文档中的节点或节点集，节点是通过沿着路径（path）或者步（steps）来选取的，使用 XPath 获取数据主要是找到数据所在的路径。示例如下：

```
<html>
 <head>
  <base href='http://example.com/' />
  <title>Example website</title>
 </head>
 <body>
  <div id='images'>
   <a href='image1.html'>Name: My image 1 <br/>
      <img src='image1 thumb.jpg'/></a>
   <a href='image2.html'>Name: My image 2 <br/>
      <img src='image2 thumb.jpg'/></a>
   <a href='image3.html'>Name: My image 3 <br/>
```

```
      <img src='image3 thumb.jpg'/></a>
   <a href='image4.html'>Name: My image 4 <br/>
      <img src='image4 thumb.jpg'/></a>
   <a href='image5.html'>Name: My image 5 <br/>
      <img src='image5 thumb.jpg'/></a>
  </div>
 </body>
</html>
```

根据上述例子获取标签<a>，href 属性为 image1.html 的内容，XPath 语法如下：

```
xpath('//div[@id="images"]/a[@href="image1.html]"/text()').extract()
```

或者：

```
xpath('//a[@href="image1.html]"/text()').extract()
```

对于上述两种不同的定位方法，说明如下：

（1）第一种方法是因为标签<a>嵌套在<div>中，所以先通过//div[@id="images"]对<div>定位，在已定位<div>的基础上添加/a[@href="image1.html]，说明先查找<div>，再查找<div>里面的<a>。

（2）第二种方法是因为标签<a>的 href 属性为 image1.html，在整段 HTML 中具有唯一性，所以直接对标签<a>定位即可。

上述例子使用“//”“/”和“@”这类特殊符号，这是 XPath 的路径表达式，常用的 XPath 路径表达式如表 22-1 所示。

表 22-1　XPath 路径表达式

表达式	描述
nodename	选取此节点的所有子节点
/	从根节点选取
//	从匹配选择的当前节点选择文档中的节点，而不考虑它们的位置
.	选取当前节点
..	选取当前节点的父节点
@	选取属性

除了路径表达式外，XPath 的方括号（[]）可嵌套谓语（用来查找某个特定的节点或者包含某个指定值的节点）。简单地说，方括号中可编写标签的特性，从而精确地找出所需的数据，如表 22-2 所示。

表 22-2　路径表达式及结果

路径表达式	结果
/div/a[1]	选取属于 div 的第一个 a 标签
/div/a[last()]	选取 div 里最后的 a 标签
/div/a[last()-1]	选取 div 里倒数第二个 a 标签
/div/a[position()<3]	选取 div 前两个 a 标签
/div/a[@id!="image1.html"]	选取 div 里属性 id 不为 image1.html 的 a 标签

XPath 的定位与 Windows 系统的路径相同，但计算机上同一目录不允许存在同名的文件或文件夹，而 HTML 可存在这种情况，为了解决这个问题，XPath 可对标签的属性进行筛选，若有多个同时符合的条件，则会获取全部符合条件的数据。

22.7 文 件 下 载

爬虫除了爬取数据之外，还常常需要爬取文件，如图片、文本文件和音视频等。Scrapy 在下载图片（文件）时提供了一个可重用的 Item Pipelines，称为 Media Pipeline。Media Pipeline 分为 Files Pipeline 和 Images Pipeline，两者实现的功能如下：

（1）能避免重新下载已下载过的数据。
（2）可以指定下载后保存的路径。

Images Pipeline 为处理图片提供了额外的功能：

（1）将所有下载的图片格式转换成普通的 JPEG 并使用 RGB 颜色模式。
（2）生成缩略图。
（3）检查图片的宽度和高度，确保它们满足最小的尺寸限制。

Pipeline 同时会在内部保存一个被调度下载的 URL 列表，然后将包含相同媒体的链接关联到这个队列上来，从而防止重复下载。

使用 Files Pipeline 实现下载的步骤如下：

步骤01 在 Spider 中爬取一个 Item 后，将相应的文件 URL 放入 file_urls 字段中。

步骤02 Item 被返回之后就会转交给 Item Pipeline。

步骤03 当这个 Item 到达 FilesPipeline 时，在 file_urls 字段中的 URL 列表会通过标准的 Scrapy 调度器和下载器来调度下载，并且优先级很高，在抓取其他页面前就被处理。而这个 Item 会一直在这个 Pipeline 中被锁定，直到所有的文件下载完成。

步骤04 当文件被下载完之后，结果会被赋值给另一个 files 字段。这个字段包含一个关于下载文件的新字典列表，比如下载路径、源地址、文件校验码。files 里面的顺序和 file_url 的顺序是一致的。若下载出错，则不会出现在这个 files 中。

ImagesPipeline 的使用跟 FilesPipeline 差不多，不过使用的字段名不一样，image_urls 用于保存图片的 URL 地址，使用 ImagesPipeline 的好处是可以通过配置来提供额外的功能，比如生成文件缩略图、通过图片大小过滤需要下载的图片等。ImagesPipeline 使用 Pillow 来生成缩略图以及转换成标准的 JPEG/RGB 格式。

为了进一步掌握 Scrapy 的下载功能，分别找出三个文件下载链接：

```
# 下载 zip 压缩包
'http://d.1.didiwl.com/PYTHON zryycl.zip',
# 下载图片
'https://www.python.org/static/img/python-logo.png',
# 下载歌曲文件
```

```
'http://124.232.176.137/mp3.9ku.com/m4a/655825.m4a'
```

然后创建新的 Scrapy 项目，名为 scrapy_download，在 CMD 窗口下创建 scrapy_download 项目，指令如下：

```
scrapy startproject scrapy_download
```

创建项目后，打开项目的 setting.py 文件，添加以下配置代码：

```
ITEM PIPELINES = {
   'scrapy_download.pipelines.ScrapyDownloadPipeline': 300,
   'scrapy download.pipelines.DownloadFlie': 1,
}
# 设置保存路径
FILES STORE = 'E:\\full\\'
# 设置请求头
DEFAULT REQUEST HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,application/xml;
             q= 0.9,*/*;q=0.8',
  'User-Agent': 'Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0)
                 Gecko/20100101 Firefox/41.0'
}
```

setting.py 除了配置请求头和文件保存路径之外，还对 ITEM_PIPELINES 添加了 DownloadFlie 类，当程序执行数据存储的时候，会同时执行 ScrapyDownloadPipeline 和 DownloadFlie 类。完成 setting.py 配置后，打开 items.py，定义 Items 类属性，代码如下：

```
import scrapy
class ScrapyDownloadItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
    FileUrl = scrapy.Field()
    FileName = scrapy.Field()
    pass
```

Items 定义了 FileUrl 和 FileName，分别是文件下载链接和文件名。接着在 spiders（文件夹）下新建 download_spider.py 文件，代码如下：

```
from scrapy download.items import ScrapyDownloadItem
from scrapy.spider import Spider
# 导入 setting.py 配置信息
from scrapy.conf import settings
class downspider(Spider):
    name = "downspider"
    allowed domains = []
    start urls = [
       'http://124.232.176.137/mp3.9ku.com/m4a/655825.m4a'
       ]

    def parse(self, response):
        # 下载方法一
        f = open(settings['FILES STORE']+'MySong.m4a', 'wb')
        f.write(response.body)
        f.close()
        # 下载方法二
```

```
        item = ScrapyDownloadItem()
        item['FileName'] = ['PythonBook.zip', 'Python.jpg', 'MyMusic.m4a']
        item['FileUrl']=['http://d.1.didiwl.com/PYTHON_zryycl.zip',
                        'https://www.python.org/static/img/python-logo.png',
                        'http://124.232.176.137/mp3.9ku.com/m4a/655825.m4a']
        return item
```

从代码中看到，函数 parse 实现了以下两种下载方式：

（1）方法一是通过访问文件链接得到链接的响应内容（文件的字节流），然后将响应内容（文件的字节流）写入文件，这种下载方式与 requests 库下载文件一致。

（2）方法二是将文件链接和文件名写入 Items 对象，然后将 Items 传到 Item Pipeline 实现文件下载。

最后在 pipelines.py 实现下载功能，代码如下：

```
from scrapy.pipelines.files import FilesPipeline
from scrapy.pipelines.images import ImagesPipeline
import scrapy
# 导入 setting.py 配置信息
from scrapy.conf import settings

class ScrapyDownloadPipeline(object):
    def process_item(self, item, spider):
        # 入库处理等操作
        return item

# 下载功能
class DownloadFlie(FilesPipeline):
    # 重写 get_media_requests
    def get_media_requests(self, item, info):
        for index, url in enumerate(item['FileUrl']):
            yield scrapy.Request(url, meta={'name': item['FileName'][index]})

    # 重写 file_path，设置下载的文件名
    def file_path(self, request, response=None, info=None):
        file_name = settings['FILES_STORE'] + (request.meta['name'])
        return file_name
```

代码中定义了类 ScrapyDownloadPipeline 和 DownloadFlie：

- ScrapyDownloadPipeline 是在创建项目时自动生成的，在此不做任何处理，但程序依然会执行，因为已被 setting.py 的 ITEM_PIPELINES 激活。
- DownloadFlie 是自定义的类，继承自父类 FilesPipeline（ImagesPipeline），然后将父类的方法 get_media_requests 和 file_path 进行重写。

get_media_requests 的说明如下：

（1）遍历 item['FileUrl']（三个文件下载链接），在 for 循环中使用 enumerate()获取下载链接所在列表的序号。

（2）获取序号对应 item['FileName']的文件名。

（3）scrapy.Request 设置了参数 meta，该参数主要给函数 file_path 传递文件名。

file_path 的说明如下：

（1）settings['FILES_STORE']用于获取 setting.py 的路径配置信息，request.meta['name']用于获取函数 get_media_requests 的 scrapy.Request()中的 meta 信息。

（2）如果不重写该方法，Scrapy 就会自行定义文件名，但 Scrapy 对文件命名存在一定缺陷，比如下载链接不规范会出现文件无法保存的情况。

在 CMD 窗口运行 scrapy_download 项目，程序运行完成后，查看文件下载情况，如图 22-11 所示。



图 22-11　Scrapy 下载文件

22.8　本 章 小 结

Scrapy 是一个为了爬取网站数据、提取结构性数据而编写的应用框架，主要应用在数据挖掘、信息处理或存储历史数据等一系列程序中。其最初是为了页面抓取所设计的，也可以应用在获取 API 所返回的数据（例如 Amazon Associates Web Services）或者通用的网络爬虫中。通过本章的学习，读者应当掌握以下技能：

1. Scrapy 的运行机制

（1）引擎从调度器中取出一个 URL（URL），用于接下来的抓取。

（2）引擎把 URL 封装成请求（Request）传给下载器，下载器把资源下载后封装成应答包（Response）。

（3）爬虫解析 Response。

（4）若解析出实体（Item），则交给实体管道进行进一步的处理。

（5）若解析出的是 URL，则把 URL 交给 Scheduler 等待抓取。

数据抓取的主要目标是从非结构化来源（通常是网页）中提取结构化数据。Scrapy 可以将提取的数据作为 Python 字典返回，但 Python 字典缺乏结构，字典的键会在输入时出现拼写错误或者返回数据不一致，因此，Scrapy 提供了 Items 对象，用于管理和规范爬取数据，使其结构规范化。

2. Item Pipeline 的编写规则

当 Spiders 爬取的数据存放到 Items 之后，回调函数的 return（yield）返回 Items 对象，就会触

发 Item Pipeline 对 Items 对象的操作，实现数据存储。Item Pipeline 的编写规则如下：

（1）setting.py 配置数据库信息。数据库信息最好在 setting.py 中配置，这符合统一规范化开发要求。

（2）对 pipelines.py 的类初始化（__init__）函数实现数据库连接。如果使用 SQLAlchemy 入库，还需创建映射类映射数据表。

（3）最后由函数 process_item 实现数据入库。

当抓取网页时，最常见的任务是从 HTML 源码中提取数据，Scrapy 提取数据有一套机制，被称作选择器（Seletors），通过特定的 XPath 或者 CSS 表达式来选择 HTML 中的某部分数据。当然，LXML 和 BeautifulSoup 也可以在 Scrapy 中担任数据清洗的角色。

3. 使用 Files Pipeline 实现下载的步骤

（1）在 Spider 中爬取一个 Item 后，将相应的文件 URL 放入 file_urls 字段中。

（2）Item 被返回之后就会转交给 Item Pipeline。

（3）当这个 Item 到达 FilesPipeline 时，在 file_urls 字段中的 URL 列表会通过标准的 Scrapy 调度器和下载器来调度下载，并且优先级很高，在抓取其他页面前就被处理。而 Item 会一直在这个 Pipeline 中被锁定，直到所有的文件下载完成。

（4）当文件被下载完之后，结果会被赋值给另一个 files 字段。这个字段包含一个关于下载文件的新字典列表，比如下载路径、源地址、文件校验码。files 里面的顺序和 file_url 的顺序是一致的。若下载出错，则不会出现在这个 files 中。

第 23 章

Scrapy 扩展开发

23.1 剖析 Scrapy 中间件

我们知道，Scrapy 框架开发爬虫程序必须遵循框架的开发规则，使爬虫程序的开发实现了规范化，提高了开发效率。Scrapy 框架的强大不仅于此，它还有很好的扩展性——自定义开发，可以满足开发人员实现不同的开发需求。

Scrapy 的自定义开发主要体现在 Scrapy 的中间件上，也就是 Scrapy 项目里的 middlewares.py 文件，该文件里以类的形式定义中间件，在配置文件 settings.py 注册即可激活中间件，当项目在运行的时候，Scrapy 会自动调用自定义中间件。

创建 Scrapy 项目的时候，middlewares.py 文件默认定义了两个中间件，分别是 SpiderMiddleware 和 DownloaderMiddleware，前者是爬虫中间件，介于 Scrapy 引擎和爬虫之间的框架，主要工作是处理爬虫的响应输入和请求输出；后者是下载器中间件，位于 Scrapy 引擎和下载器之间的框架，处理引擎与下载器之间的请求及响应。两者的工作流程如下：

（1）Scrapy 向爬取网站发送 HTTP 请求由中间件 DownloaderMiddleware 的 process_request() 函数实现。

（2）请求发送成功后，调用 DownloaderMiddleware 的 process_response()函数生成相应的响应内容，并将响应内容发送给 Scrapy 引擎。

（3）Scrapy 引擎将响应内容传递给 SpiderMiddleware 的 process_spider_input()函数处理，根据开发者编写的 Spider 程序来对响应内容进行清洗处理。

（4）SpiderMiddleware 将处理结果返回给 Scrapy 引擎，这个过程是由 process_spider_output() 函数实现。而 Scrapy 引擎再将结果传递给 Item Pipeline，从而实现数据存储。

23.1.1 SpiderMiddleware 中间件

在 middlewares.py 文件里查看中间件 SpiderMiddleware 的定义方式，该中间件定义了 6 个方法，每个方法负责实现不同的功能，具体的代码如下：

```
from scrapy import signals
class MySpiderMiddleware(object):
    # Not all methods need to be defined. If a method is not defined,
    # scrapy acts as if the spider middleware does not modify the
    # passed objects.
    @classmethod
    def from crawler(cls, crawler):
        # This method is used by Scrapy to create your spiders.
        s = cls()
        crawler.signals.connect(s.spider_opened,
                                signal=signals.spider opened)
        return s

    def process spider input(self, response, spider):
        # Called for each response that goes through the spider
        # middleware and into the spider.
        # Should return None or raise an exception.
        return None

    def process spider output(self, response, result, spider):
        # Called with the results returned from the Spider, after
        # it has processed the response.
        # Must return an iterable of Request, dict or Item objects.
        for i in result:
            yield i

    def process spider exception(self, response, exception, spider):
        # Called when a spider or process spider input() method
        # (from other spider middleware) raises an exception.
        # Should return either None or an iterable of Response, dict
        # or Item objects.
        pass

    def process start requests(self, start requests, spider):
        # Called with the start requests of the spider, and works
        # similarly to the process spider output() method, except
        # that it doesn't have a response associated.
        # Must return only requests (not items).
        for r in start requests:
            yield r

    def spider opened(self, spider):
        spider.logger.info('Spider opened: %s' % spider.name)
```

中间件 SpiderMiddleware 以类的形式表示，默认情况下，类名为“项目名+SpiderMiddleware”，而自定义中间件可根据开发者的喜好自行命名。上述代码的 6 个方法说明如下：

- from_crawler()是访问 settings 和 signals 的入口函数，比如读取配置文件 settings.py 的某些

配置。重写这个方法需要返回实例对象；如果返回的对象里带有参数，可在初始化方法__init__()使用，如下所示：

```
class MySpiderMiddleware(object):
    # 重写 __init__
    # 参数 MySetting 来自 from_crawler 的返回参数
    def __init__(self,MySetting=None)
        pass
    # 重写 from_crawler
    @classmethod
    def from_crawler(cls, crawler):
        # 从配置文件读取 MySetting 内容
        MySetting=crawler.settings.get('MySetting')
        # 返回实例对象，并设置参数 MySetting
        return cls(MySetting)
```

- process_spider_input()是处理网页的响应内容，如果方法的返回值为 None，则代表该方法执行完成。参数 response 代表网页的响应内容；参数 spider 代表项目 spider 文件夹的 spider 程序。
- process_spider_output()是将响应内容的处理结果返回到 spider 程序。参数 response 代表网页的响应内容；参数 result 代表响应内容的处理结果；参数 spider 代表项目 spider 文件夹的 spider 程序。
- process_spider_exception()是在 spider 程序或 process_spider_input()方法引发异常时执行的处理。参数 response 代表当前请求的响应内容；参数 exception 代表异常对象，包含异常信息；参数 spider 代表项目 spider 文件夹的 spider 程序。
- process_start_requests()在 spider 程序发送 HTTP 请求的时候调用。参数 start_requests 代表 HTTP 请求对象，参数 spider 代表项目 spider 文件夹的 spider 程序。
- spider_opened()是 spider 程序的运行记录，参数 spider 代表项目 spider 文件夹的 spider 程序。

在 middlewares.py 文件里定义的中间件还需要在配置文件 settings.py 里注册激活才能使用，打开配置文件 settings.py，并找到配置属性 SPIDER_MIDDLEWARES。Scrapy 项目创建的时候，该属性已被注释，只要将注释去掉即可，如果项目里自定义了中间件，在该配置属性下写入自定义中间件即可生效，如下所示：

```
SPIDER_MIDDLEWARES = {
# 注册激活项目默认的中间件 MyScrapySpiderMiddleware
'MyScrapy.middlewares.MyScrapySpiderMiddleware': 543,
# 注册激活自定义中间件 MySpiderMiddleware
'MyScrapy.middlewares.MySpiderMiddleware': 100,
}
```

配置属性 SPIDER_MIDDLEWARES 以字典格式表示，字典的 key 代表中间件的路径信息，如“MyScrapy”代表文件夹 MyScrapy（即项目名），“middlewares”代表 middlewares.py 文件，“MyScrapySpiderMiddleware”代表文件 middlewares.py 里定义的 MyScrapySpiderMiddleware 类。

字典的 value 代表中间件执行的优先级别，数值越低，优先级越高，比如上述代码激活了两个中间件，当 Scrapy 引擎使用中间件 SpiderMiddleware 的时候，则优先执行中间件 MySpiderMiddleware，最后再执行中间件 MyScrapySpiderMiddleware。

在 Scrapy 框架里提供了 5 个内置的 SpiderMiddleware 中间件，这些中间件的源码可在 Python 的安装目录 Lib\site-packages\scrapy\spidermiddlewares 下找到，它们实现的功能说明如表 23-1 所示。

表 23-1 SpiderMiddleware 内置中间件及其功能说明

SpiderMiddleware 中间件	功能说明
DepthMiddleware	追踪每个请求在爬取网站的深度位置
HttpErrorMiddleware	过滤所有失败或错误的 HTTP 状态码，可节省计算机的资源消耗
OffsiteMiddleware	当前请求的 URL 域名不符合 spider 的 allowed_domains，则过滤该请求
RefererMiddleware	根据请求的响应内容来设置请求头的 Referer 字段
UrlLengthMiddleware	若当前请求的 URL 不符合中间件所设置的 URL 长度，则过滤该请求

若想在 Scrapy 项目里使用内置中间件，必须了解每个中间件的定义方式，比如 UrlLengthMiddleware，该中间件需要在配置文件 settings.py 中设置属性 URLLENGTH_LIMIT，用于判断 URL 的长度是否符合要求。此外，中间件还需要在配置属性 SPIDER_MIDDLEWARES 中进行注册激活，各个内置中间件的注册激活方法如下：

```
{
'scrapy.contrib.spidermiddleware.httperror.HttpErrorMiddleware': 50,
'scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware': 500,
'scrapy.contrib.spidermiddleware.referer.RefererMiddleware': 700,
'scrapy.contrib.spidermiddleware.urllength.UrlLengthMiddleware': 800,
'scrapy.contrib.spidermiddleware.depth.DepthMiddleware': 900,
}
```

23.1.2 DownloaderMiddleware 中间件

在 middlewares.py 文件里查看中间件 DownloaderMiddleware 的定义方式，该中间件定义了 5 个方法，每个方法负责实现不同的功能，具体的代码如下：

```
class MyDownloaderMiddleware(object):
    # Not all methods need to be defined. If a method is not defined,
    # scrapy acts as if the downloader middleware does not modify the
    # passed objects.

    @classmethod
    def from crawler(cls, crawler):
        # This method is used by Scrapy to create your spiders.
        s = cls()
        crawler.signals.connect(s.spider opened,
                                signal=signals.spider opened)
        return s

    def process request(self, request, spider):
        # Called for each request that goes through the downloader
        # middleware.
        # Must either:
        # - return None: continue processing this request
        # - or return a Response object
        # - or return a Request object
```

```
        # - or raise IgnoreRequest: process exception() methods of
        #   installed downloader middleware will be called
        return None

    def process response(self, request, response, spider):
        # Called with the response returned from the downloader.
        # Must either;
        # - return a Response object
        # - return a Request object
        # - or raise IgnoreRequest
        return response

    def process exception(self, request, exception, spider):
        # Called when a download handler or a process request()
        # (from other downloader middleware) raises an exception.
        # Must either:
        # - return None: continue processing this exception
        # - return a Response object: stops process exception() chain
        # - return a Request object: stops process exception() chain
        pass

    def spider opened(self, spider):
        spider.logger.info('Spider opened: %s' % spider.name)
```

中间件 DownloaderMiddleware 以类的形式表示，默认情况下，类名为“项目名+DownloaderMiddleware”，自定义中间件可根据开发者的喜好自行命名。上述代码的 5 个方法说明如下：

- from_crawler()是访问 settings 和 signals 的入口函数，它的作用与中间件 SpiderMiddleware 的一致。
- process_request()是 Scrapy 发送 HTTP 请求时所调用的方法。参数 request 代表当前请求对象，如请求头、请求方式和请求 URL 等信息；参数 spider 代表项目 spider 文件夹的 spider 程序。方法返回值可为 None、Response 对象、Request 对象或 IgnoreRequest 对象，各个返回值的说明如下。
  - None：这是常见的返回值，代表方法执行完成并往下执行爬虫程序。
  - Response 对象：停止 process_request()的执行，并开始执行 process_response()。
  - Request 对象：停止当前中间件的执行，将当前的请求给 Scrapy 引擎重新执行。
  - IgnoreRequest 对象：抛出 Scrapy 定义的异常对象，再由 process_exception()处理异常，而当前请求不再往下执行。
- process_response()是生成当前请求的响应内容，并将响应内容发送给 Scrapy 引擎。参数 request 代表当前请求对象，带有请求头、请求方式和请求地址等信息；参数 response 是当前请求的响应内容；参数 spider 代表项目 spider 文件夹的 spider 程序。方法返回值可为 Response 对象、Request 对象或 IgnoreRequest 对象，各个返回值的说明如下。
  - Response 对象：将响应内容传递给其他中间件的 process_response()或 Scrapy 引擎。
  - Request 对象：停止当前中间件的执行，将当前的请求给 Scrapy 引擎重新执行。
  - IgnoreRequest 对象：抛出 Scrapy 定义的异常对象，再由 process_exception()处理异常，而当前请求不再往下执行。

- process_exception()是在 spider 程序、process_request()或 process_response()方法引发异常时而执行的处理。参数 request 代表引发异常时所对应的响应内容；参数 exception 代表异常对象，包含异常信息；参数 spider 代表项目 spider 文件夹的 spider 程序。

spider_opened()是 spider 程序的运行记录，参数 spider 代表项目 spider 文件夹的 spider 程序。

中间件 DownloaderMiddleware 的注册激活方式与中间件 SpiderMiddleware 相同，只不过配置文件的配置属性有所不同，其配置属性为 DOWNLOADER_MIDDLEWARES。

在 Scrapy 框架里提供了 15 个内置的 DownloaderMiddleware 中间件，这些中间件的源码可在 Python 的安装目录 Lib\site-packages\scrapy\downloadermiddlewares 下找到，它们实现的功能说明如表 23-2 所示。

表 23-2　DownloaderMiddleware 内置中间件及其说明

DownloaderMiddleware 中间件	功能说明
CookiesMiddleware	设置当前请求的 Cookies 信息
DefaultHeadersMiddleware	当前请求设置默认请求头（即配置属性 DEFAULT_REQUEST_HEADERS）
DownloadTimeoutMiddleware	设置当前请求的超时时间
HttpAuthMiddleware	为当前请求完成 HTTP 认证过程
HttpCacheMiddleware	为整个请求提供底层（low-level）缓存支持
HttpCompressionMiddleware	提供了对压缩（gzip, deflate）数据的支持
ChunkedTransferMiddleware	提供分块传输编码功能
HttpProxyMiddleware	设置当前请求的 HTTP 代理
RedirectMiddleware	根据响应内容的状态码处理重定向的请求
MetaRefreshMiddleware	根据响应内容的 HTML 的 meta-refresh 标签处理重定向的请求
RetryMiddleware	对错误或失败的请求进行重新的调度，再次发送 HTTP 请求
RobotsTxtMiddleware	根据网站的 robots.txt 来筛选可爬取的网页内容
DownloaderStats	保存其他中间件的方法信息
UserAgentMiddleware	设置当前请求的请求头的 User-Agent 字段
AjaxCrawlMiddleware	根据响应内容的 HTML 的 meta-fragment 标签查找 AJAX 可爬取页面

若想在 Scrapy 项目里使用内置中间件，必须了解每个中间件的定义方式。此外，中间件还需要在配置属性 DOWNLOADER_MIDDLEWARES 进行注册激活，各个内置中间件的注册激活方法如下：

```
    {
    'scrapy.contrib.downloadermiddleware.robotstxt.RobotsTxtMiddleware': 100,
    'scrapy.contrib.downloadermiddleware.httpauth.HttpAuthMiddleware': 300,
    'scrapy.contrib.downloadermiddleware.downloadtimeout.DownloadTimeoutMiddle
ware': 350,
    'scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware': 400,
    'scrapy.contrib.downloadermiddleware.retry.RetryMiddleware': 500,
    'scrapy.contrib.downloadermiddleware.defaultheaders.DefaultHeadersMiddlewa
re': 550,
    'scrapy.contrib.downloadermiddleware.redirect.MetaRefreshMiddleware': 580,
    'scrapy.contrib.downloadermiddleware.httpcompression.HttpCompressionMiddle
ware': 590,
```

```
    'scrapy.contrib.downloadermiddleware.redirect.RedirectMiddleware': 600,
    'scrapy.contrib.downloadermiddleware.cookies.CookiesMiddleware': 700,
    'scrapy.contrib.downloadermiddleware.httpproxy.HttpProxyMiddleware': 750,
    'scrapy.contrib.downloadermiddleware.chunked.ChunkedTransferMiddleware':
830,
    'scrapy.contrib.downloadermiddleware.stats.DownloaderStats': 850,
    'scrapy.contrib.downloadermiddleware.httpcache.HttpCacheMiddleware': 900,
    }
```

总的来说，中间件 SpiderMiddleware 和 DownloaderMiddleware 与 spider 程序是紧密关联的。spider 程序是通过中间件与 Scrapy 引擎进行交互，也就是说，中间件是作为 spider 程序与 Scrapy 引擎的通信桥梁，而通过自定义中间件可有效改变通信方式，从而满足开发需求。

23.2 自定义中间件

虽然 Scrapy 内置了多个中间件，但大多数情况下，开发者更乐意编写自定义中间件，这样不仅能满足开发需求，而且可节省开发时间。如果使用内置中间件，开发者需要对内置中间件有一定的掌握，并且还可能需要重写内置中间件，这样对比自定义中间件来说，它的开发周期相对较长。

中间件的自定义是实现类的定义，该类可选择是否继承某个父类，比如定义 MyDownloaderMiddleware 中间件，该中间件可以继承 objects 类（Python 的新式类），还可以继承 Scrapy 的内置中间件，使得自定义中间件具有内置中间件的功能。

在自定义中间件里可以定义多个属性、方法以及重写初始化方法等操作，但是要让中间件起到实际作用，某些方法是必不可少的。比如 process_request()或 process_response()方法等，我们只要重写这些方法，就能起到自定义作用，因为 Scrapy 引擎根据这些方法名来执行处理，若自定义中间件没有自定义这些方法，Scrapy 引擎会按照原有的方法执行，这样就失去了自定义的意义。

23.2.1 设置代理 IP 服务

在爬虫开发过程中，使用代理 IP 向网站发送 HTTP 请求是有效解决反爬虫的方法之一。虽然 Scrapy 内置了 HTTP 代理的中间件 HttpProxyMiddleware，但分析 HttpProxyMiddleware 的源码得知，它实现的功能与我们常用的代理 IP 功能有所不同，因此代理 IP 功能只能通过自定义中间件的方式实现。

首先在 CMD 命令提示符窗口下创建 Scrapy 项目，项目名为 useProxy，并且在项目的 spiders 文件夹里创建 proxySpider.py 文件，整个项目的目录结构如图 23-1 所示。

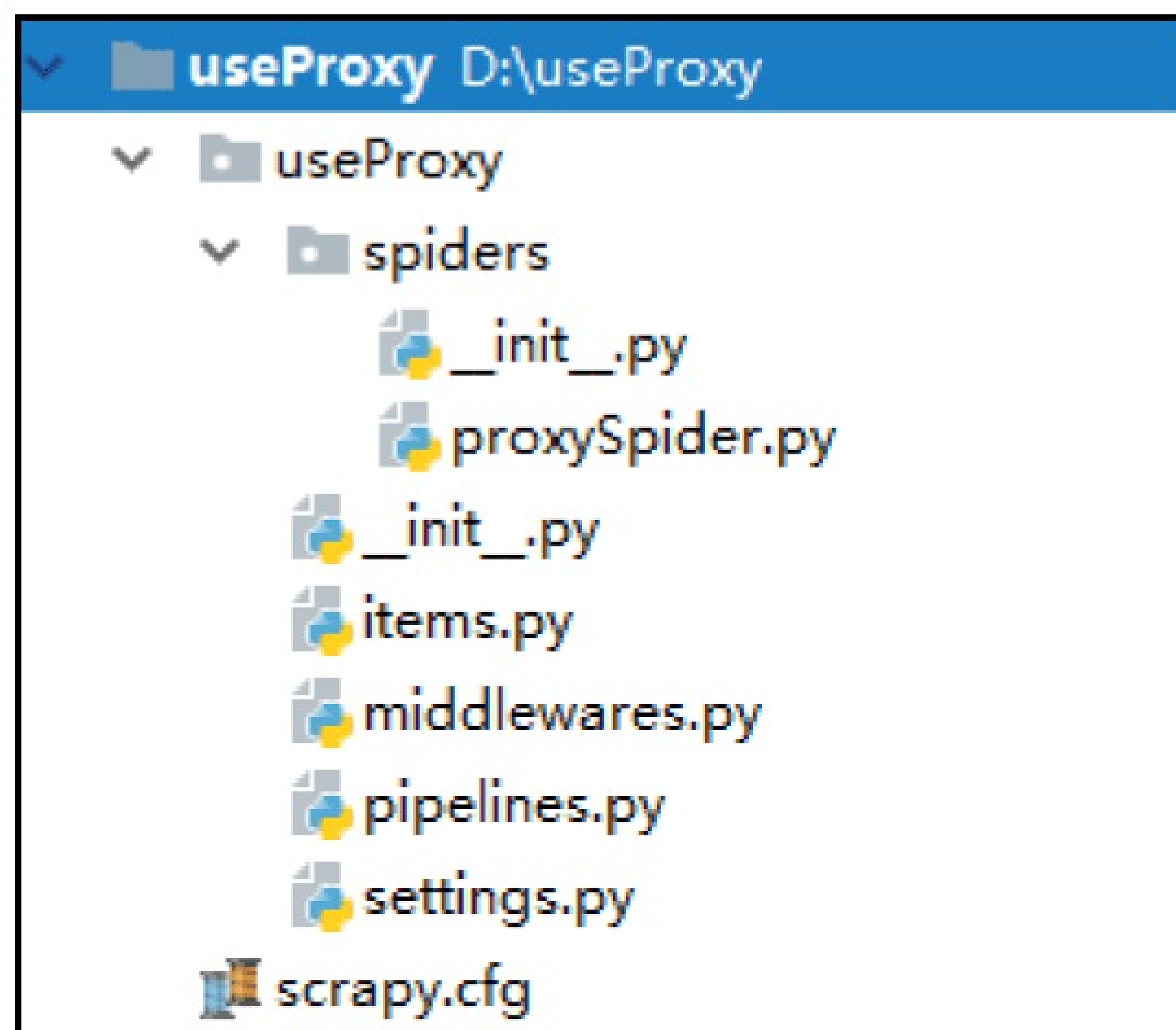


图 23-1　useProxy 的目录结构

打开项目的 middlewares.py 文件，文件已有的代码不做任何修改，并在最下方的空白位置自定义中间件 HttpbinProxyMiddleware。自定义中间件的代码如下：

```
# 自定义中间件
import requests
class HttpbinProxyMiddleware(object):
    def process_request(self, request, spider):
        url = 'http://api.ip.data5u.com/socks/get.html?
                 order=66e943bad6ca54b010168351e5f53186&
                 json=1&type=1&sep=3'
        pro_addr = requests.get(url).json().get('data', '')
        if pro_addr:
            ip = pro_addr[0].get('ip')
            port = pro_addr[0].get('port')
            request.meta['proxy']='http://'+str(ip)+':'+str(port)
```

中间件 HttpbinProxyMiddleware 定义了方法 process_request()，这是处理 Scrapy 向网站发送 HTTP 请求信息。在该方法里，使用 Python 的 requests 模块向第三方代理 IP 服务平台发送 HTTP 请求，从而获取代理 IP 的地址；再将代理 IP 的地址以参数 proxy 的形式写入 Scrapy 的当前请求，使 Scrapy 的当前请求以代理 IP 的形式向网站发送 HTTP 请求。

在配置文件 settings.py 里注册激活自定义中间件，自定义中间件属于 DownloaderMiddleware 类型，因此将配置属性 DOWNLOADER_MIDDLEWARES 的注释去除，并将自定义中间件写入配置信息里。此外，在配置属性 DEFAULT_REQUEST_HEADERS 添加 User-Agent 以及配置属性 ROBOTSTXT_OBEY 设置为 False。参见如下代码：

```
ROBOTSTXT_OBEY = False
# 注册激活中间件
DOWNLOADER_MIDDLEWARES = {
   'useProxy.middlewares.UseproxyDownloaderMiddleware': 543,
   'useProxy.middlewares.HttpbinProxyMiddleware': 543,
}
# 添加 User-Agent
DEFAULT_REQUEST_HEADERS = {
  'Accept':'text/html,application/xhtml+xml,
```

```
    application/xml;q=0.9,*/*;q=0.8',
      'Accept-Language': 'en',
      'User-Agent':'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                    AppleWebKit/537.36 (KHTML, like Gecko)
                    Chrome/69.0.3497.100 Safari/537.36'
    }
```

最后在 proxySpider.py 文件里编写爬虫程序，验证代理 IP 服务是否正常使用。本次爬取的网站是一个 HTTP 测试网站，该网站返回用户的请求信息，根据网站返回的信息来验证代理 IP 服务。在浏览器上访问“http://httpbin.org/get”网址，在网页中找到当前本地的外网 IP 地址，如图 23-2 所示。

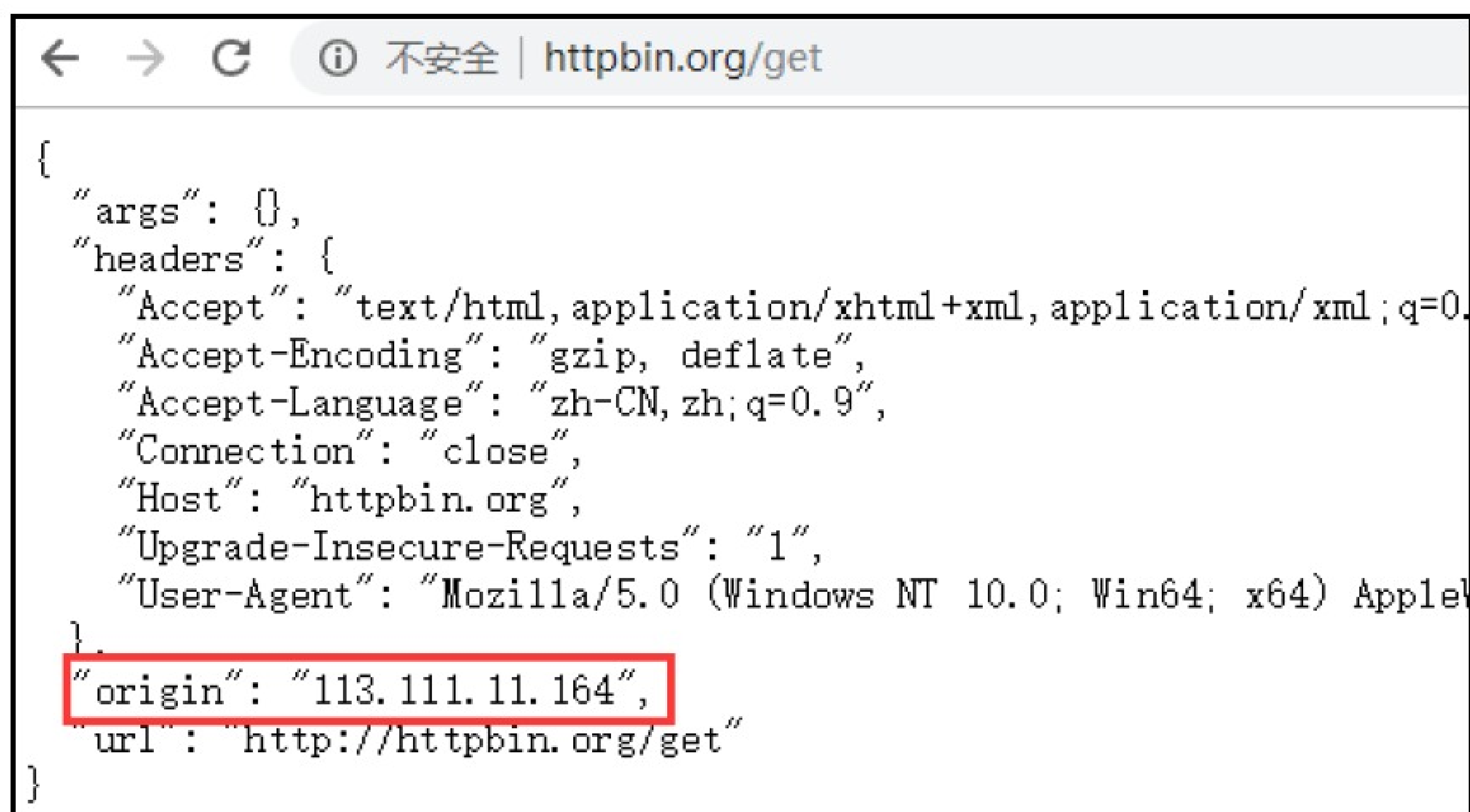


图 23-2　本地的外网 IP 地址

在 proxySpider.py 文件编写 spider 程序 HttpbinSpider()，将 HTTP 测试网站的 URL 作为爬取对象，并将网站的响应内容输出。spider 程序的代码如下：

```
import scrapy
class HttpbinSpider(scrapy.Spider):
    name = "httpbin"
    allowed domains = ["httpbin.ort/get"]
    start urls = ['http://httpbin.org/get']
    def parse(self, response):
        print(response.text)
```

至此，整个代理 IP 的中间件 HttpbinProxyMiddleware 开发已完成。在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl httpbin，启动并运行项目 useProxy。在项目运行的日志中可以找到 HTTP 测试网站的响应内容，如图 23-3 所示。



图 23-3　项目 useProxy 的运行结果

在上述例子中，中间件 HttpbinProxyMiddleware 是对所有的 HTTP 请求都使用代理 IP 访问。在实际的开发中，需要根据不同的 URL 来合理选择代理 IP，可在中间件 HttpbinProxyMiddleware 判断当前请求的 URL 域名，从而执行相应的处理，如下所示：

```
import requests
class HttpbinProxyMiddleware(object):
    def process_request(self, request, spider):
        # 判断 URL 的域名是否使用代理 IP
        if 'google.com' in request.url:
            url = 'http://api.ip.data5u.com/socks/get.html?
                    order=66e943bad6ca54b010168351e5f53186&
                    json=1&type=1&sep=3'
            pro_addr=requests.get(url).json().get('data','')
            if pro_addr:
                ip = pro_addr[0].get('ip')
                port = pro_addr[0].get('port')
                request.meta['proxy']='http://'+str(ip)+':'+str(port)
        else:
            return None
```

23.2.2　动态设置请求头

在分析网站的时候，每个请求信息的请求头属性都会各不相同，还有一些网站会根据请求头内容来制定一系列的反爬虫策略。对于 Scrapy 框架来说，请求头可以在 spider 程序里定义并以参数 headers 的形式传递给 HTTP 请求，如下所示：

```
import scrapy
from scrapy.spider import Request
class HeaderSpider(scrapy.Spider):
    name = "headers"
    allowed_domains = ["httpbin.ort/get"]
    start_urls = ['http://httpbin.org/get']
    def start_requests(self):
        self.headers = {
            'User-Agent': 'Mozilla/4.0 (compatible; MSIE 7.0;
                           Windows NT 5.1; 360SE)'
        }
```

```
        yield Request(url=self.start_urls[0],headers=self.headers,
                      callback=self.parse)
    def parse(self, response):
        print(response.text)
```

上述例子是为当前 HTTP 请求设置特定的请求头，若一个 spider 程序里有多个不同的 HTTP 请求，那么每个请求的请求头有可能出现内容重叠，这样很容易造成代码的冗余。对于这些重复定义或者交互调用的情况，可以将请求头以中间件的形式实现。

在 CMD 命令提示符窗口下创建 Scrapy 项目，项目名为 requestHeader，并且在项目的 spiders 文件夹里创建 headerSpiders.py 文件，整个项目的目录结构如图 23-4 所示。

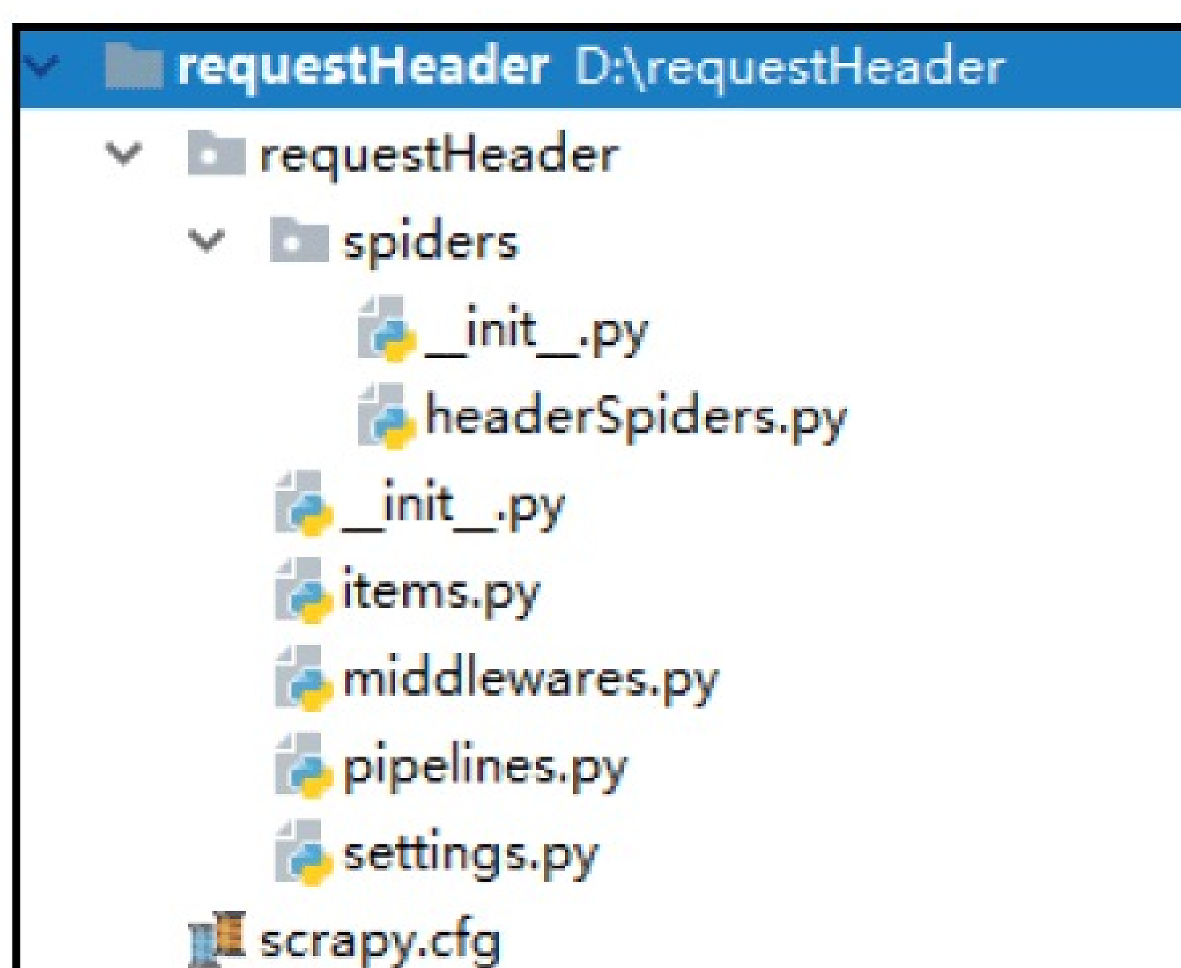


图 23-4　requestHeader 的目录结构

打开项目的 middlewares.py 文件，文件已有的代码不做任何修改，并在最下方的空白位置自定义中间件 HeaderMiddleware。自定义中间件的代码如下：

```
# 根据参数 meta 的 requsetHeader 动态设置请求头
class HeaderMiddleware(object):
    def process_request(self, request, spider):
        header = request.meta.get('requsetHeader','')
        if header:
            for key, values in header.items():
                request.headers.setdefault(key,values)
```

中间件 HeaderMiddleware 定义了方法 process_request()，在该方法里，获取当前请求 request 的参数 meta 的属性 requsetHeader，参数 meta 是由 spider 程序设置，属性 requsetHeader 的值以字典的形式表示，字典的 key 代表请求头的属性，字典的 value 代表请求头的属性值。遍历字典的键值对，将每个键值对作为当前请求的请求头内容。

在配置文件 settings.py 里注册激活自定义中间件，自定义中间件是属于 DownloaderMiddleware 类型，因此将配置属性 DOWNLOADER_MIDDLEWARES 的注释去除，并将自定义中间件写入配置信息里。此外，在配置属性 DEFAULT_REQUEST_HEADERS 添加 User-Agent 以及配置属性 ROBOTSTXT_OBEY 设置为 False。如下所示：

```
ROBOTSTXT_OBEY = False
# 注册激活中间件
DOWNLOADER_MIDDLEWARES = {
    'requestHeader.middlewares.HeaderMiddleware': 300,
    'requestHeader.middlewares.RequestheaderDownloaderMiddleware': 543,
```

```
}
# 添加 User-Agent
DEFAULT REQUEST HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,
             application/xml;q=0.9,*/*;q=0.8',
  'Accept-Language': 'en',
  'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                AppleWebKit/537.36 (KHTML, like Gecko)
                Chrome/69.0.3497.100 Safari/537.36'
}
```

最后在 headerSpiders.py 文件里编写爬虫程序，爬取的网站还是 HTTP 测试网站，根据网站返回的响应内容来验证请求头的设置是否正确。spider 程序代码如下所示：

```
import scrapy
from scrapy.spider import Request
class HeaderSpider(scrapy.Spider):
    name = "headers"
    allowed domains = ["httpbin.ort/get"]
    start urls = ['http://httpbin.org/get']
    def start requests(self):
        headers = {'Referer': 'https://www.baidu.com/',
                   'Upgrade-Insecure-Requests': '1',
                   'Accept-Language': 'zh-CN,zh;q=0.9'}
        yield Request(url=self.start urls[0],
                      meta={'requsetHeader': headers},
                      callback=self.parse)
    def parse(self, response):
        print(response.text)
```

spider 程序定义了字典 headers，并将字典传入到参数 meta 的 requsetHeader。字典 headers 是设置请求头的可变属性，比如当前请求需要使用 Referer、Upgrade-Insecure-Requests 和 Accept-Language 属性。而请求头的公用属性设置在配置文件 settings.py 的 DEFAULT_REQUEST_HEADERS 属性。

至此，整个请求头的中间件 HeaderMiddleware 开发已完成。在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl headers，启动并运行项目 requestHeader。在项目运行的日志中找到 HTTP 测试网站的响应内容，如图 23-5 所示。

```
"headers": {
  "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8",
  "Accept-Encoding": "gzip,deflate,br",
  "Accept-Language": "zh-CN,zh;q=0.9",
  "Connection": "close",
  "Host": "httpbin.org",
  "Referer": "https://www.baidu.com/",
  "Upgrade-Insecure-Requests": "1",
  "User-Agent": "Scrapy/1.5.1 (+https://scrapy.org)"
},
"origin": "113.111.11.164",
"url": "http://httpbin.org/get"
```

图 23-5 项目 requestHeader 的运行结果

23.2.3　设置随机 Cookies

Cookies 在网络爬虫里担任着一个重要的角色，它代表当前用户信息，当某些网页设有用户访问权限的时候，在爬虫里只需将用户登录后的 Cookies 信息加入 HTTP 请求即可解决访问权限的问题。Cookies 不仅能代表用户信息，还能制定反爬虫策略，比如通过 Cookies 内容构建 HTTP 请求的请求参数及 Cookies 的动态更新等，具体的反爬虫策略会在后续的章节详细介绍。

对于 Scrapy 框架来说，Cookies 信息可在 Spider 程序里定义并以参数 cookies 的形式传递给 HTTP 请求，代码如下：

```
import scrapy
from scrapy.spider import Request
class cookiesSpider(scrapy.Spider):
    name = "cookies"
    allowed domains = ["httpbin.ort/get"]
    start urls = ['http://httpbin.org/get']
    def start requests(self):
        cookies = {'MyCookies': 'hello Python'}
        yield Request(url=self.start urls[0],cookies=cookies,
                              callback=self.parse)
    def parse(self, response):
        print(response.text)
```

上述例子是为当前 HTTP 请求设置 Cookies 信息，若 Spider 程序有多个 HTTP 请求，而且每个请求的 URL 地址需要使用不同的 Cookies。每次发送 HTTP 请求的时候，都需要自定义相关的 Cookies 内容，无形之中造成代码冗余，不利于日后的维护和管理，因此，我们可以将 Cookies 以中间件的形式表示。

在 CMD 命令提示符窗口下创建 Scrapy 项目，项目名为 useCookies，并且在项目的 spiders 文件夹里创建 cookiesSpiders.py 文件，整个项目的目录结构如图 23-6 所示。

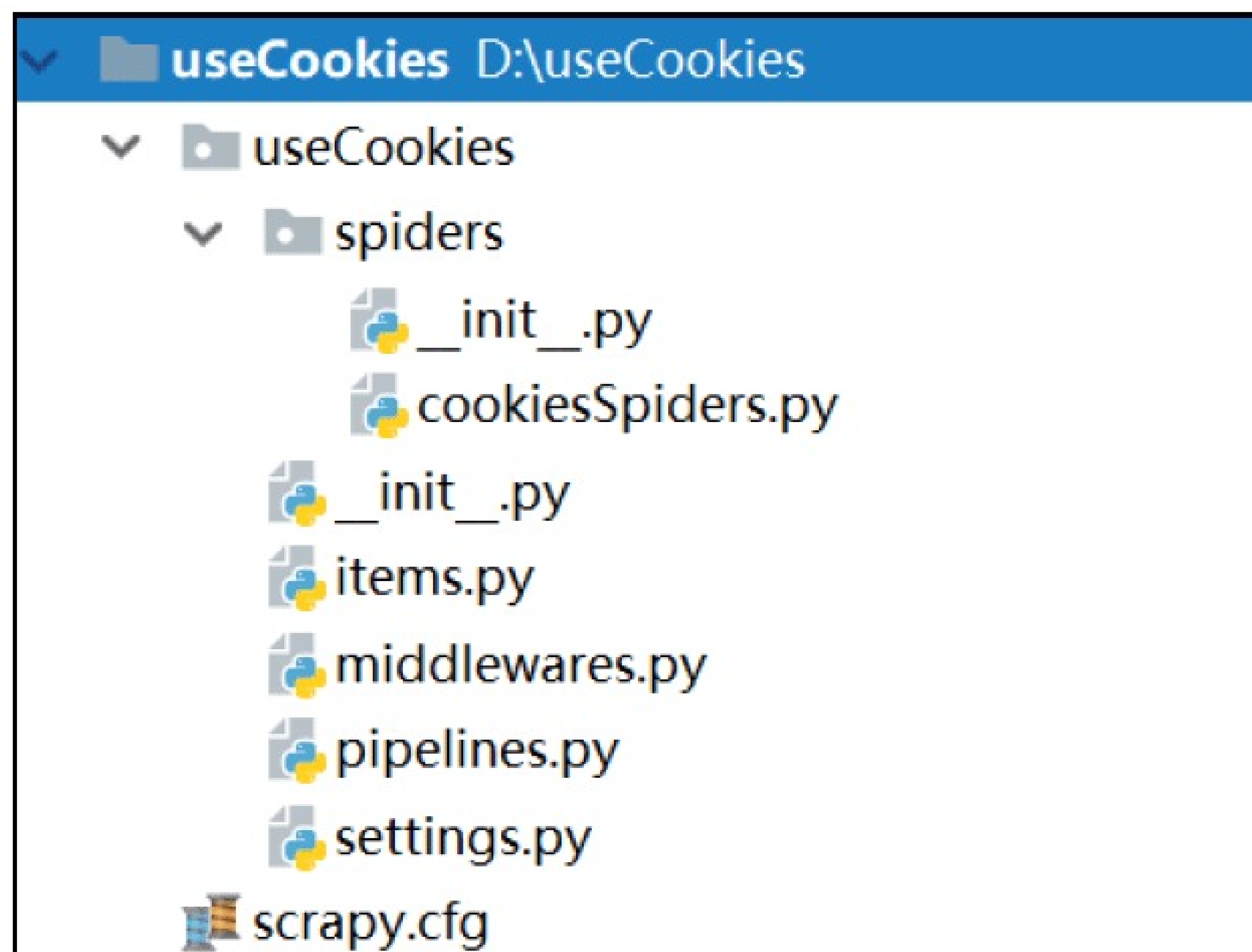


图 23-6　useCookies 的目录结构

打开项目的 middlewares.py 文件，文件已有的代码不做任何修改，并在最下方的空白位置自定义中间件 CookiesMiddleware。自定义中间件的代码如下：

```
import random
class CookiesMiddleware(object):
    def __init__(self, cookies):
        self.cookies = cookies

    @classmethod
    def from_crawler(cls, crawler):
        obj = cls(
            cookies=crawler.settings.get('COOKIES_LIST'),
        )
        return obj

    def process_request(self, request, spider):
        cookie = random.choice(self.cookies)
        request.cookies = cookie
```

中间件 CookiesMiddleware 重写初始化方法和定义方法 from_crawler()和 process_request()。中间件实现的功能说明如下：

- 重写初始化方法是为中间件设置初始化参数 cookies，并将 cookies 的参数值赋值给属性 cookies。
- 类方法 from_crawler()是在中间件初始化的时候，为初始化参数 cookies 提供参数值，参数值来自配置文件 settings.py 的配置属性 COOKIES_LIST。
- 配置属性 COOKIES_LIST 的属性值以列表表示，每个列表元素代表网站的 Cookies 信息并以字典的形式表示。
- 在方法 process_request()里，通过 Python 的随机函数 random 来对配置属性 COOKIES_LIST 进行随机选取，并将选中的 Cookies 传入当前的 HTTP 请求里。

在配置文件 settings.py 里注册激活自定义中间件，自定义中间件属于 DownloaderMiddleware 类型，因此将配置属性 DOWNLOADER_MIDDLEWARES 的注释去除，并将自定义中间件写入配置信息里。此外，将配置属性 ROBOTSTXT_OBEY 设置为 False 及设置配置属性 COOKIES_LIST。代码如下：

```
ROBOTSTXT_OBEY = False
# 注册激活中间件
DOWNLOADER_MIDDLEWARES = {
   'useCookies.middlewares.CookiesMiddleware': 300,
   'useCookies.middlewares.UsecookiesDownloaderMiddleware': 543,
}
# 设置 Cookies 列表
COOKIES_LIST = [
    {'MyCookies': 'hello Python'},
    {'YourCookies': 'hi Python'},
    {'ItsCookies': 'hello World'}
]
```

最后在 cookiesSpiders.py 文件里编写爬虫程序，爬取的网站还是 HTTP 测试网站，根据网站返回的响应内容来验证 Cookies 的设置是否正确。Spider 程序代码如下：

```
import scrapy
from scrapy.spider import Request
```

```
class cookiesSpider(scrapy.Spider):
    name = "cookies"
    allowed_domains = ["httpbin.ort/get"]
    start_urls = ['http://httpbin.org/get']
    def start_requests(self):
        yield Request(url=self.start_urls[0],callback=self.parse)
    def parse(self, response):
        print(response.text)
```

至此，整个 Cookies 的中间件 CookiesMiddleware 开发已完成。在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl cookies，启动并运行项目 useCookies。在项目运行的日志中找到 HTTP 测试网站的响应内容，如图 23-7 所示。

```
"headers": {
  "Accept": "text/html,application/xhtml+xml,application/xml;q
  "Accept-Encoding": "gzip,deflate,br",
  "Accept-Language": "en",
  "Connection": "close",
  "Cookie": "ItsCookies=hello World",
  "Host": "httpbin.org",
  "User-Agent": "Scrapy/1.5.1 (+https://scrapy.org)"
},
```

图 23-7　项目 useCookies 的运行结果

23.3　实战：Scrapy+Selenium 爬取豆瓣电影评论

Scrapy 不仅能自定义中间件，还可以将中间件结合其他模块实现不同的爬取方式。在 Scrapy 框架上使用 Selenium 模块实现爬虫开发是常见的手段之一，因为 Selenium 可以模拟用户访问浏览器，从中爬取目标数据，实现过程较为简单，而且能绕开各种反爬虫策略。本节将讲述如何在 Scrapy 框架里使用 Selenium 实现豆瓣电影评论的爬取。

23.3.1　网站分析

在开发爬虫程序之前，需要对网站进行详细分析，根据分析结果制定开发流程。在浏览器上打开某个电影评论页面（https://movie.douban.com/subject/26425063/comments?status=P），并在谷歌开发者工具分析网页内容，如图 23-8 所示。



图 23-8 网页分析

我们需要在开发者工具的 Elements 标签中分析网页，因为 Selenium 获取的网页内容与 Elements 标签的网页内容相同。分析得知，当前页面的所有电影评论是在一个属性 id 为 comments 的 div 标签里；每个评论是在一个属性 class 为 comment 的 div 标签里，而评论内容是以 span 标签嵌入在 div 标签里。

当单击网页最下方的“后页”即可访问第二页的评论内容，发现第二页的 URL 地址多了三个请求参数。为了理解这三个参数的作用，我们切换不同的页数，找出这些参数的变化规律，如图 23-9 所示。

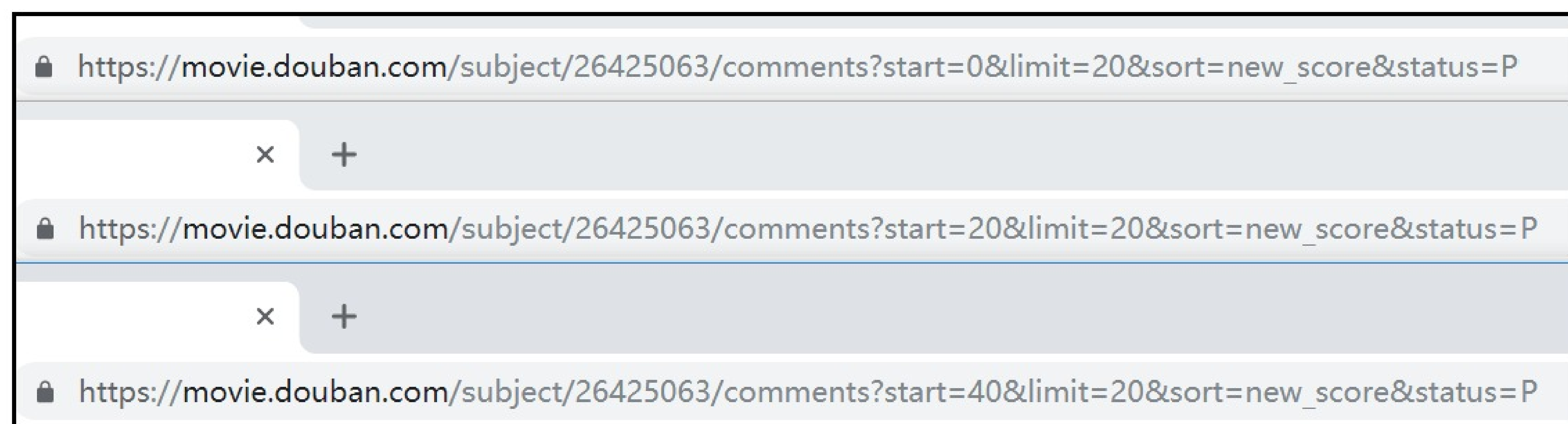


图 23-9 分析请求参数

图 23-9 上的 URL 地址从上至下排列，第一条 URL 是第一页评论；第二条 URL 是第二页评论；第三条 URL 是第三页评论，从三条 URL 的变化规律得知，URL 的构造说明如下：

- start 是当前页数的首条评论在全部评论的位置，每页的评论有 20 条，第一页的首条评论从 0 开始，第二页的首条评论从 20 开始……以此类推，得出参数 start 的值为 p × 20，p 为每页的页数，并且页数从 0 开始计算。

- limit 是每一页的评论数量，由于每页的评论数量固定为 20 条，因此该参数值为 20。
- sort 是所有评论的排序方式，参数值 new_score 是以评论的热门程序进行排序，若无特殊要求，可将该参数视为固定参数。
- status 代表当前评论的用户已经看过该电影，若无特殊要求，也可将该参数视为固定参数。
- URL 地址的“26425063”代表当前电影的 ID，只要将不同电影的 ID 替换到 URL 地址里，就可以爬取不同电影的评论内容。

23.3.2　项目设计与实现

根据网站分析设计 Scrapy 项目，在 CMD 命令提示符窗口下创建 Scrapy 项目，项目名为 douban，并且在项目 settings.py 的路径下创建 conf.ini 文件夹以及在 spiders 文件夹里创建 movie.py 文件，整个项目的目录结构如图 23-10 所示。

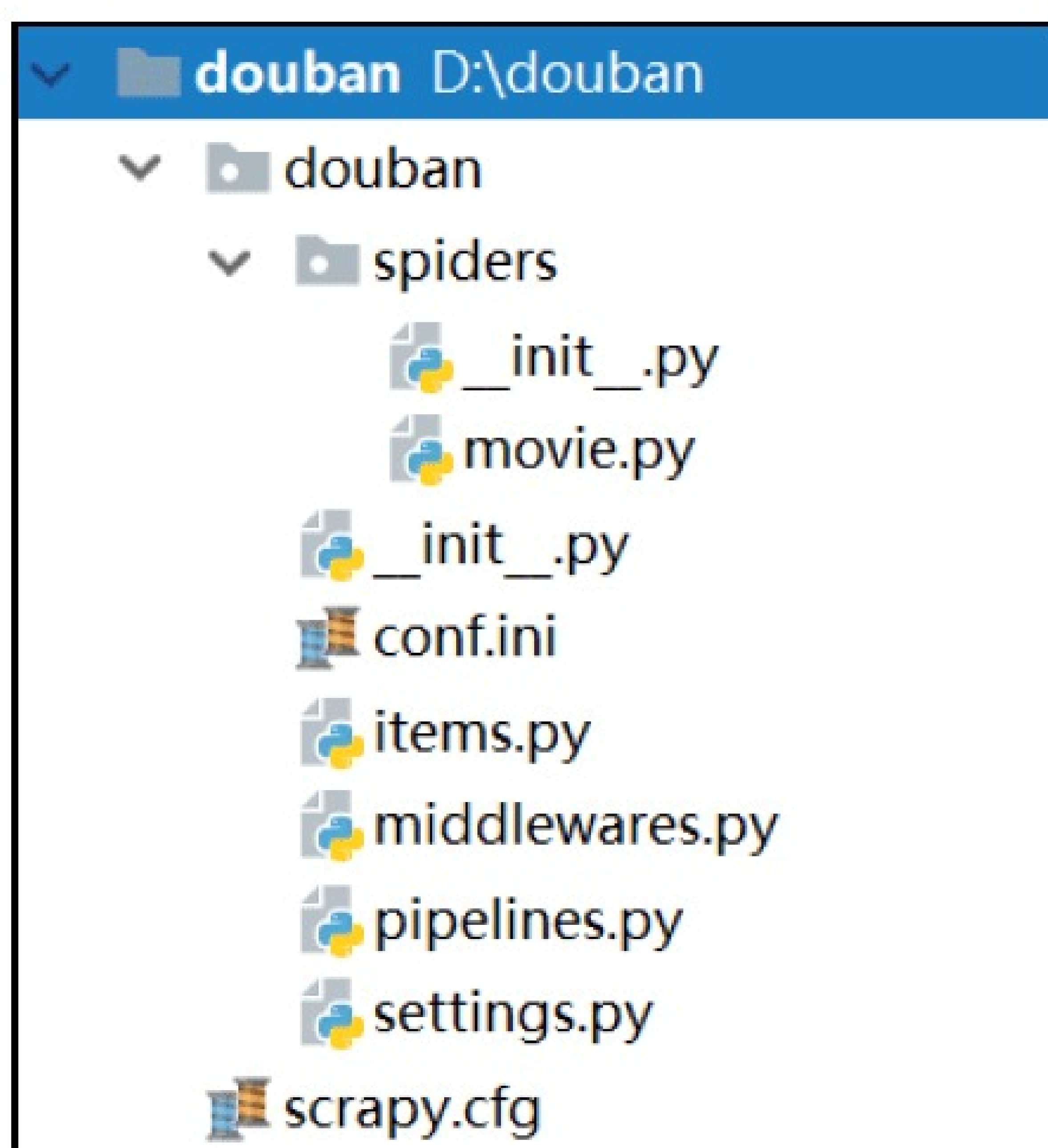


图 23-10　douban 的目录结构

项目功能分为三大部分：基本功能、Selenium 中间件开发和 spider 程序开发。基本功能由 items.py、pipelines.py 和 settings.py 实现，items.py 和 pipelines.py 实现数据存储功能，将电影评论存储在 MySQL 数据库里，存储过程由 SQLAlchemy 完成；settings.py 实现整个项目的功能配置，包括中间件的注册、请求头的设置及存储功能 ITEM_PIPELINES 等。

打开配置文件 settings.py，将项目的功能及相关配置写入文件，文件的代码如下：

```
BOT_NAME = 'douban'
SPIDER_MODULES = ['douban.spiders']
NEWSPIDER_MODULE = 'douban.spiders'
# ROBOTSTXT_OBEY 改为 False
ROBOTSTXT_OBEY = False

DEFAULT_REQUEST_HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,
             application/xml;q=0.9,*/*;q=0.8',
  'Accept-Language': 'en',
  # 加上 User-Agent，否则提示 403 错误信息
```

```
      'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                     AppleWebKit/537.36 (KHTML, like Gecko) Chrome
                     /69.0.3497.100 Safari/537.36'
    }
    # 注册自定义中间件 SeleniumMiddleware
    DOWNLOADER_MIDDLEWARES = {
       'douban.middlewares.DoubanDownloaderMiddleware': 543,
       'douban.middlewares.SeleniumMiddleware': 300,
    }
    # 注册管道，开启数据存储功能
    ITEM_PIPELINES = {
       'douban.pipelines.DoubanPipeline': 300,
    }
    # 设置 Selenium 的超时时间
    SELENIUM_TIMEOUT = 30
    import os
    # 设置配置文件 conf.ini 路径信息
    BASE_DIR = os.path.dirname(os.path.realpath(__file__))
    CONF = os.path.join(BASE_DIR, 'conf.ini')
    # 设置数据库连接信息
    MYSQL_CONNECTION = 'mysql+pymysql://root:1234@
    localhost/spiderdb?charset=utf8mb4'
```

配置文件 settings.py 主要用于修改功能配置及设置基本信息：修改功能配置是对项目已有的配置属性进行修改，比如 DEFAULT_REQUEST_HEADERS 和 DOWNLOADER_MIDDLEWARES 等；设置基本信息是为项目新增一些配置属性，如新增配置文件 conf.ini 和数据库连接信息。

> **注　意**
>
> 数据库连接编码使用 utf8mb4 是为了保证数据能完全录入数据库，因为电影评论里可能出现某些特殊字符，比如手机特有的表情，这些表情的编码格式已超出 utf8 的编码范围，所以只能选择 utf8mb4，数据库 spiderdb 在创建时也要选择 utf8mb4 编码。

打开项目的 items.py 文件，将项目需要存储的字段在此文件进行定义，我们定义了两个字段，分别是电影 ID 和电影评论内容，如下所示：

```
    import scrapy
    class DoubanItem(scrapy.Item):
        movieId = scrapy.Field()
        comment = scrapy.Field()
        pass
```

根据已定义的字段，在项目的 pipelines.py 文件编写相关的数据存储过程。文件里定义了两个类，分别代表数据表映射类 scrapy_db 和数据存储 DoubanPipeline，具体的定义方式如下：

```
    from sqlalchemy import *
    from sqlalchemy.orm import sessionmaker
    from sqlalchemy.ext.declarative import declarative_base
    # 导入 setting 配置信息
    from scrapy.conf import settings

    # 定义映射类
```

```
    Base = declarative base()
    class scrapy db(Base):
         tablename    = 'douban db'
       id = Column(Integer(), primary_key=True)
       movieId = Column(String(100))
       comment = Column(String(2000))

    class DoubanPipeline(object):
       def   init  (self):
          # 初始化，连接数据库
          conntion = settings['MYSQL CONNECTION']
          engine = create_engine(conntion,echo=False,pool_size=2000)
          DBSession = sessionmaker(bind=engine)
          self.SQLsession = DBSession()
          # 创建数据表
          Base.metadata.create all(engine)

       def process item(self, item, spider):
          # 入库处理
          self.SQLsession.execute(scrapy_db.__table__.insert(),
                  {'comment': item['comment'].replace("\n", ""),
                   'movieId': item['movieId']})
          self.SQLsession.commit()
          return item
```

数据存储 DoubanPipeline 通过重写初始化方法__init__()，在 DoubanPipeline()实例化的时候，读取配置文件 settings.py 的数据库连接信息，并由 SQLAlchemy 实现数据库连接；方法 process_item()是将 Scrapy 引擎传递的参数 item 写入数据库并保存。

23.3.3　定义 Selenium 中间件

配置文件 settings.py 已注册自定义中间件 SeleniumMiddleware，因此在项目的 middlewares.py 文件里定义中间件 SeleniumMiddleware，该中间件是将 Scrapy 的 HTTP 请求改为由 Selenium 模块实现，代码如下：

```
    # 自定义 Selenium
    from selenium import webdriver
    from selenium.webdriver.chrome.options import Options
    from selenium.webdriver.support.ui import WebDriverWait
    from scrapy.http import HtmlResponse

    class SeleniumMiddleware(object):
       def   init  (self, timeout=None):
          self.timeout=timeout
          self.chrome options=Options()
          self.chrome options.add argument('--headless')
          self.driver=webdriver.Chrome(chrome options=self.chrome options)
          self.wait=WebDriverWait(self.driver, self.timeout)

       def process request(self, request, spider):
           # 参数 usedSelenium 决定是否使用 Selenium
          if request.meta['usedSelenium']:
```

```
            try:
                # 生成一个页面的 driver 对象
                self.driver.get(request.url)
                # 直接返回响应内容
                return HtmlResponse(url=request.url,
                                    body=self.driver.page_source,
                                    request=request,encoding='utf-8',
                                    status=200)
            except:
                  # 若出现异常，抛出 HTTP 状态码 500
                return HtmlResponse(url=request.url,status=500,
                                    request=request)
         # 如不使用 Selenium，则执行原有的访问方式
        else:
            return None

    def __del__(self):
        self.driver.close()

    @classmethod
    def from_crawler(cls, crawler):
         # 读取 settings.py 的 SELENIUM_TIMEOUT
         # 用于初始化方法的实例化
        return cls(timeout=crawler.settings.get('SELENIUM_TIMEOUT'),)
```

中间件 SeleniumMiddleware 共定义了 4 个方法，核心方法是 process_request()，其他的方法是为核心方法提供相关数据和设置，具体说明如下：

- from_crawler() 是类方法，该方法是从配置文件 settings.py 里读取配置属性 SELENIUM_TIMEOUT，它为初始化方法__init__()的参数 timeout 提供具体的参数值。
- __del__()是 Python 内置的删除方法，这是对象在销毁前所执行的方法，可理解为中间件的清理方法，它将 Selenium 生成的浏览器进行关闭处理。
- __init__()是中间件 SeleniumMiddleware 的初始化方法，它是将 Selenium 进行实例化，并对实例化对象配置相关属性，如浏览器无头模式和超时时间等。
- process_request()是根据当前 HTTP 请求的参数 usedSelenium 进行判断，如果参数为真，则将当前请求改为 Selenium 访问，若访问过程中出现异常就抛出 HTTP 500，这是代表当前请求失败；如果参数为假，则当前请求就按照 Scrapy 原有的方式执行。

23.3.4 开发 Spider 程序

项目的 spider 程序是在项目的 movie.py 文件里实现，从网站分析得知，电影评论页的 URL 地址带有电影 ID，只要切换不同的 ID 就能爬取不同电影的评论。我们将电影 ID 写入配置文件 conf.ini，再由 spider 程序读取并实现不同电影的评论爬取。按此设计，spider 程序的代码如下：

```
from douban.items import DoubanItem
from scrapy.selector import Selector
from scrapy.spider import Spider, Request
import configparser
```

```
class MovieSpider(Spider):
    # 属性 name 必须设置，而且是唯一命名，用于运行爬虫
    name = "Movie"
    # 设置允许访问域名
    allowed domains = ["https://movie.douban.com"]
    # 设置 URL
    start urls = 'https://movie.douban.com/subject/%s/comments?
                start=%s&limit=20&sort=new score&status=P'
    # 重写 start_requests
    def start requests(self):
        # 读取配置文件，获取电影 ID 并生成列表
        conf = configparser.ConfigParser()
        urlsList = []
        conf.read(self.settings.get('CONF'))
        temp = conf['config']
        if 'movieId' in temp.keys():
            urlsList=conf['config']['movieId'].split(',')

        for u in urlsList:
            # 根据电影 ID 选择爬取方式
            if str(u) in '26425063':
                value = True
            else:
                value = False
            # 每部电影爬取两页的评论
            for page in range(2):
                url = self.start_urls %(str(u), str(page * 20))
                yield Request(url=url,meta={'movieId':str(u),
                         'usedSelenium': value},callback=self.parse)

    def parse(self, response):
        # 将响应内容生成 Selector，用于数据清洗
        sel = Selector(response)
        # 定义 DoubanItem 对象
        item = DoubanItem()
        comments=sel.xpath('//div[@id="comments"]//div[@class="comment"]')
        for c in comments:
            item['movieId'] = response.meta['movieId']
            item['comment'] = ''.join(c.xpath('.//p//span//
                             text()').extract()).strip()
            yield item
```

在上述的 spider 程序里，电影评论页的 URL 地址以类属性 start_urls 表示，并且设置两个动态的变量，分别是电影 ID 和评论页数，这样可以爬取不同电影的不同页数的评论。spider 程序重写 start_requests()和 parse()方法，两者实现的功能说明如下：

- start_requests()读取配置文件 conf.ini 来获取电影 ID，并将电影 ID 以列表表示，每个列表元素代表了一部电影的 ID。
- 将列表进行遍历处理，如果当前列表元素（电影 ID）为“26425063”，则参数 usedSelenium 设为 True，中间件 SeleniumMiddleware 就会使用 Selenium 访问当前的 URL 地址。反之，则将 URL 地址按 Scrapy 原有的方式执行。

- 每部电影进行两次遍历，这两次遍历是构建同一部电影而不同页数的 URL 地址，把不同页数的 URL 地址和相关参数发送给 Scrapy 引擎，Scrapy 引擎根据相关参数调度相应的中间件，从而实现 HTTP 请求。
- 当中间件完成 HTTP 请求并将响应结果交给 Scrapy 引擎后，再由 Scrapy 引擎调度 parse() 方法，从而实现数据存储。

从 spider 程序读取配置文件 conf.ini 的方式可以知道，每个电影 ID 是以英文逗号进行拼接，然后赋值给配置属性 movieId，如图 23-11 所示。



图 23-11 配置信息

在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl Movie，启动并运行项目 douban。项目运行完成后，打开 spiderdb 数据库查看 douban_db 数据表的数据信息，如图 23-12 所示。

对象 | douban_db @spiderdb (My...

开始事务 文本 筛选 排序 导入 导出

id	movieId	comment
1	26636712	第一个彩蛋比正片好看？
2	26636712	女主妈30年不晕不染眼影睫毛膏了解一下~
3	26636712	1.把路易斯、蚁人、钢铁侠、星爵、死侍、蜘蛛侠废除超能力后关在一个房间里，请问
4	26636712	娱乐性很强，可惜反派太弱，好在一看就是给妇联4过渡用的。蚁人那哥们和死侍还有
5	26636712	妈妈实在太美了......
6	26636712	MARVEL為了不讓蟻人出現在婦聯4裡也是費盡心機啊…
7	26636712	量子空间是开美容院的好地方。
8	26636712	漫威用了一整部电影来解释蚁人为什么没有在复联3中出现，详情请见第一个彩蛋。
9	26636712	如果忽略第一个彩蛋的话，这就是一部很轻松、很合家欢的电影。几段追车戏真的很

图 23-12 电影评论内容

23.4 实战：Scrapy+Splash 爬取 B 站动漫信息

Scrapy 框架也可以与 Splash 模块结合使用，从 Scrapy 框架结构可知，使用 Splash 比 Selenium 更有优势，因为 Splash 是一个异步框架，它与 Scrapy 框架能完美结合。

23.4.1　Scrapy_Splash 实现原理

scrapy_splash 是 Scrapy 的功能扩展包，它为开发者提供了 Splash 中间件及相应的 HTTP 请求方式。在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入 pip 安装指令 pip install scrapy-splash，等待安装完成。然后在 Python 安装目录下查看 scrapy_splash（Lib\site-packages\scrapy_splash）源码文件，如图 23-13 所示。

软件 (D:) › Python › Lib › site-packages › scrapy_splash ›

名称	修改日期	类型
__pycache__	2018/10/16 17:49	文件夹
__init__.py	2018/10/16 17:49	Python File
cache.py	2018/10/16 17:49	Python File
cookies.py	2018/10/16 17:49	Python File
dupefilter.py	2018/10/16 17:49	Python File
middleware.py	2018/10/16 17:49	Python File
request.py	2018/10/16 17:49	Python File
response.py	2018/10/16 17:49	Python File
responsetypes.py	2018/10/16 17:49	Python File
utils.py	2018/10/16 17:49	Python File

图 23-13　源码文件

scrapy_splash 共有 5 个功能模块，分别是数据缓存 cache.py、用户 Cookies 信息 cookies.py、中间件 middleware.py、HTTP 请求方式 request.py 和响应内容 response.py。各个功能模块说明如下：

- cache.py 定义 SplashAwareFSCacheStorage 类，并继承 Scrapy 的 FilesystemCacheStorage 缓存类，由此可见，scrapy_splash 的缓存功能是在 Scrapy 原有的缓存功能上进行修改。
- cookies.py 定义了多个 Cookies 格式转换函数，比如将 HAR 格式转化成字典格式。
- middleware.py 分别定义了一个 SpiderMiddleware 中间件和两个 DownloaderMiddleware 中间件，这是 scrapy_splash 的核心功能。
- request.py 定义 HTTP 的 GET 请求和 POST 请求，以 SplashRequest 和 SplashFormRequest 类表示，这是改变 Scrapy 原有的 HTTP 请求方式。
- response.py 定义响应内容的数据格式，以 SplashTextResponse 和 SplashJsonResponse 类表示，这也是改变 Scrapy 原有的 HTTP 响应方式。

大致了解 scrapy_splash 的原理后，接下来通过一个简单的项目来掌握如何使用 scrapy_splash 模块实现爬虫开发。

23.4.2　网站分析

在开发爬虫程序之前，需要对网站进行详细分析，根据分析结果制定相应的开发流程。在浏

览器上打开 B 站已完结动画列表（https://www.bilibili.com/v/anime/finish/#/all/default/0/1/），并利用开发者工具分析网页内容，如图 23-14 所示。



图 23-14　网页分析

Splash 获取网页的响应内容与 Selenium 的一致，因此在开发者工具的 Elements 标签里分析网页内容即可。从网页结构分析得知，当前网页的所有动画信息在属性 class 为 vd-list-cnt 的 div 标签里，而每部动画信息是在 li 标签里。

在网页的下方设置分页功能，当单击第二页和第三页的时候，网页的 URL 地址随之变化，发现当前页数与 URL 最末端的数字相互对应，如图 23-15 所示。

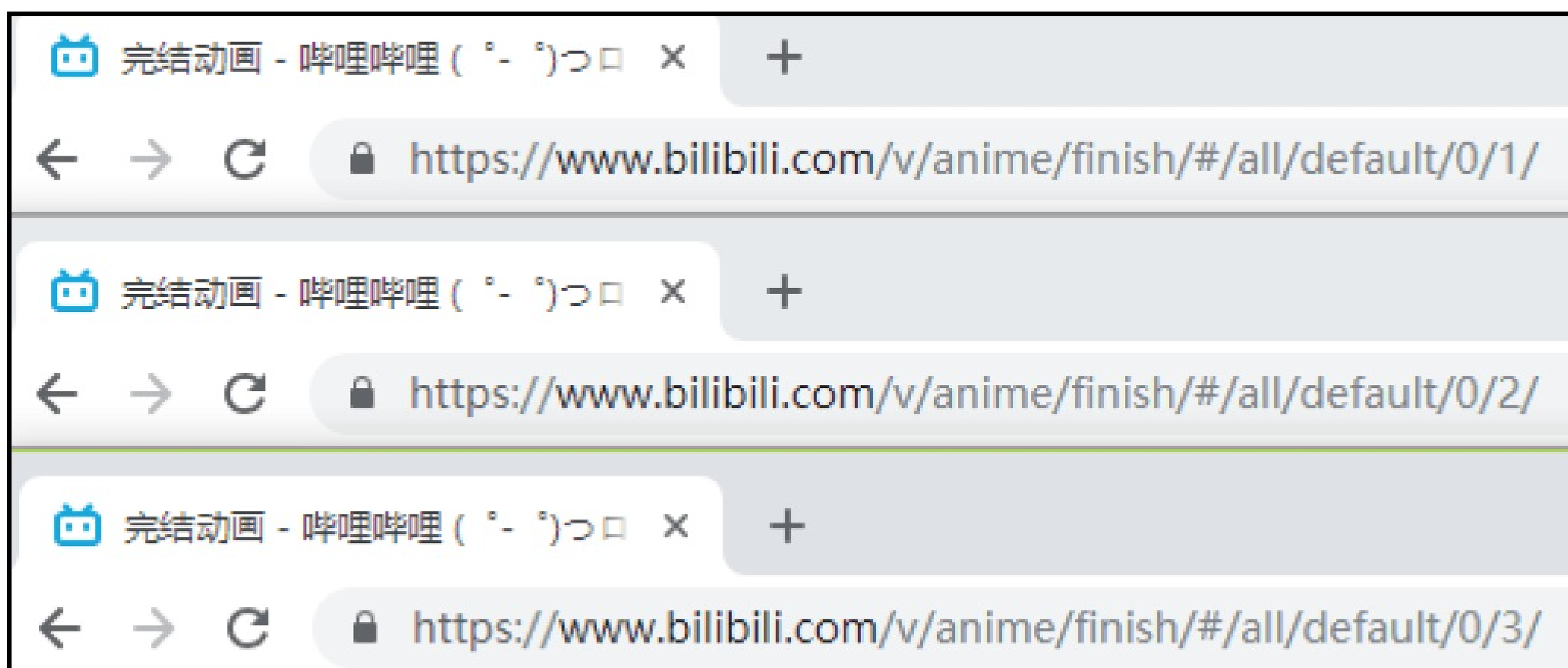


图 23-15　URL 变化规律

综合上述分析，只要将 URL 最末端的数字设为动态数据，通过设置不同的数据即可得到不同的网页内容，然后清洗网页信息并提取动画信息。

23.4.3　项目设计与实现

根据上述分析进行 Scrapy 项目设计，在 CMD 命令提示符窗口下创建 Scrapy 项目，项目名为 dongman，并且在 spiders 文件夹里创建 finishOpera.py 文件，整个项目的目录结构如图 23-16 所示。

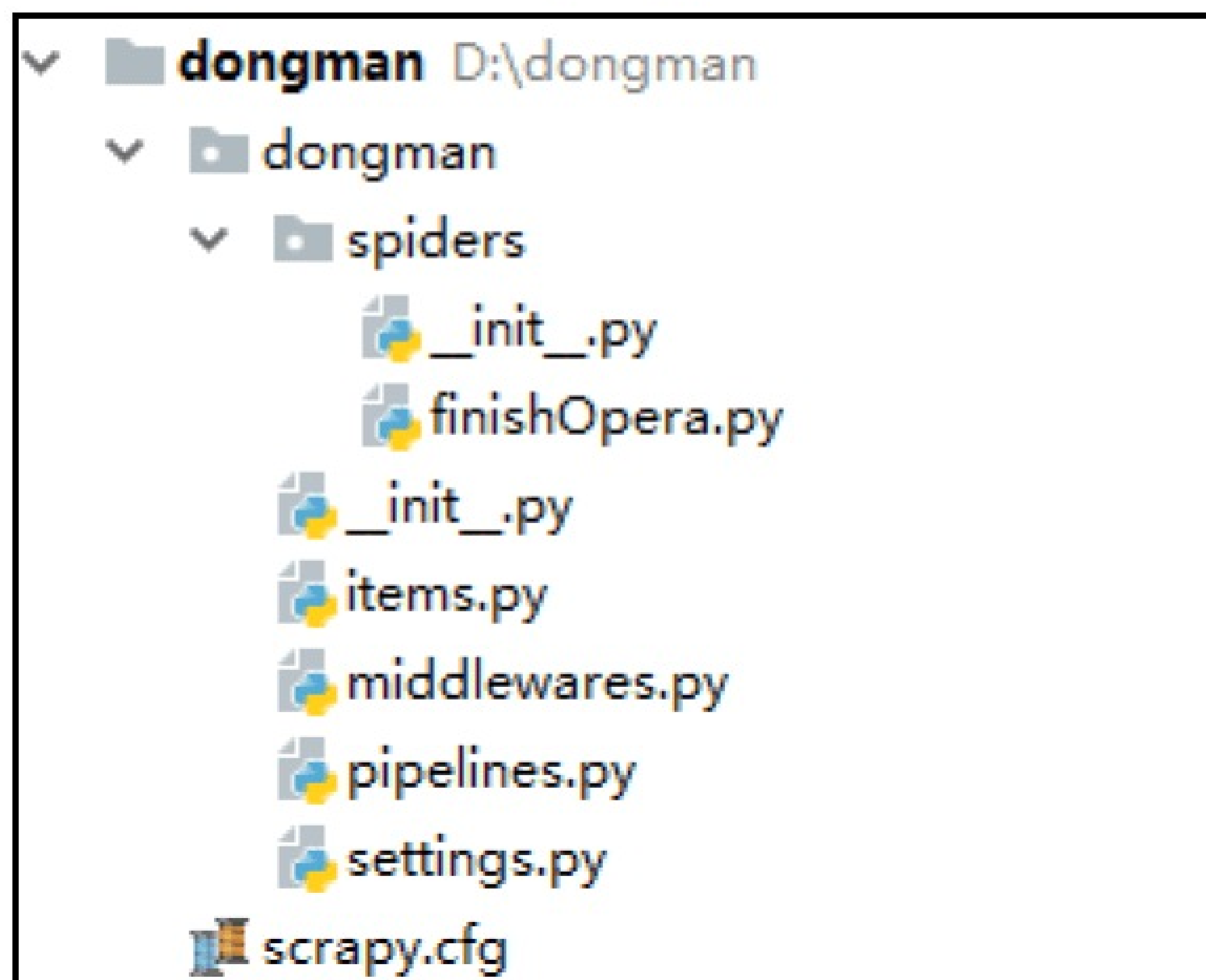


图 23-16　dongman 的目录结构

整个项目功能分为两部分：基本功能和 spider 程序开发。基本功能是由 items.py、pipelines.py 和 settings.py 实现：items.py 和 pipelines.py 实现数据存储功能，将动画信息存储在 MySQL 数据库里，存储过程由 SQLAlchemy 完成；settings.py 实现整个项目的功能配置，包括 Splash 中间件的注册、Splash 配置属性及存储功能 ITEM_PIPELINES 等。

打开配置文件 settings.py，将项目的功能以及相关配置写入文件，文件的代码如下所示：

```
BOT NAME = 'dongman'
SPIDER MODULES = ['dongman.spiders']
NEWSPIDER MODULE = 'dongman.spiders'
# 改为 False
ROBOTSTXT OBEY = False
# 注册 scrapy_splash 的中间件
SPIDER MIDDLEWARES = {
   'scrapy_splash.SplashDeduplicateArgsMiddleware': 100,
   # 'dongman.middlewares.DongmanSpiderMiddleware': 543,
}
# 注册 scrapy_splash 和 scrapy 内置的中间件
DOWNLOADER MIDDLEWARES = {
    'scrapy splash.SplashCookiesMiddleware': 723,
    'scrapy splash.SplashMiddleware': 725,
    'scrapy.downloadermiddlewares.httpcompression.
HttpCompressionMiddleware': 810,
}
# 设置 scrapy splash 配置属性
SPLASH URL = 'http://192.168.99.100:8050/'
# 去重过滤器
DUPEFILTER_CLASS='scrapy_splash.SplashAwareDupeFilter'
# 自定义 HTTP 缓存机制
HTTPCACHE_STORAGE='scrapy_splash.SplashAwareFSCacheStorage'
```

```
# 设置 Cookies，记录所有请求的发送和接收 Cookies
SPLASH COOKIES DEBUG = True
# 数据库连接信息
MYSQL CONNECTION = 'mysql+pymysql://root:1234@
                    localhost/spiderdb?charset=utf8mb4'
ITEM PIPELINES = {
   'dongman.pipelines.DongmanPipeline': 300,
}
```

配置文件 settings.py 主要是注册 scrapy_splash 的三个中间件及设置相关属性，每个配置的属性值是来自 scrapy_splash 的源码文件，如配置属性 DUPEFILTER_CLASS，它的属性值是源码文件 dupefilter.py 的 SplashAwareDupeFilter 类。

接着打开项目的 items.py 文件，将项目需要存储的字段在此文件进行定义。本项目爬取每部动画的名字、简介、观看人数和弹幕数量，分别对应的字段为 name、desc、viewNumber 和 captionsNum，如下所示：

```
import scrapy
class DongmanItem(scrapy.Item):
    name = scrapy.Field()
    desc = scrapy.Field()
    viewNumber = scrapy.Field()
    captionsNum = scrapy.Field()
```

最后在项目的 pipelines.py 文件编写相关的数据存储过程。我们定义两个类，分别代表数据表映射类 scrapy_db 和数据存储 DongmanPipeline，具体的定义方式如下：

```
from sqlalchemy import *
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative base
from scrapy.conf import settings

# 定义映射类
Base = declarative base()
class scrapy db(Base):
      tablename    = 'dongman db'
    id = Column(Integer(), primary key=True)
    name = Column(String(100))
    desc = Column(String(2000))
    viewNumber = Column(String(50))
    captionsNum = Column(String(50))

class DongmanPipeline(object):
    def   init   (self):
        # 初始化，连接数据库
        conntion = settings['MYSQL CONNECTION']
        engine = create engine(conntion,echo=False,pool size=2000)
        DBSession = sessionmaker(bind=engine)
        self.SQLsession = DBSession()
        # 创建数据表
        Base.metadata.create all(engine)

    def process item(self, item, spider):
        # 入库处理
```

```
        self.SQLsession.execute(scrapy db.  table  .insert(),
                         {'name': item['name'],
                          'desc': item['desc'],
                           'viewNumber': item['viewNumber'],
                           'captionsNum': item['captionsNum']})
        self.SQLsession.commit()
        return item
```

23.4.4　开发 Spider 程序

从网站分析得知，已完结动画的 URL 最末端的数字代表页数，只要切换不同的数值就能爬取不同页面的动画信息。Spider 程序需要将不同的 URL 地址交给 scrapy_splash 访问并获取相应的网页内容，实现代码如下：

```
from scrapy import Spider
from scrapy splash import SplashRequest
from dongman.items import DongmanItem
from scrapy.selector import Selector

class SplashSpider(Spider):
    name = 'finish opera'
    start urls = 'https://www.bilibili.com/v/anime/
                 finish/#/all/default/0/%s/'

    # 将 Scrapy 的 request 改为 SplashRequest
    def start requests(self):
        for page in range(2):
            url = self.start urls %(str(page+1))
            headers = {
                'User-Agent': 'Mozilla/5.0 (Windows NT 10.0;
                        Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
                        Chrome/69.0.3497.100 Safari/537.36'
            }
            # 设置请求信息
            args = {
                'wait': '3',
                "headers": headers,
                # 设置 Cookies
                # "cookies": {'hello': 'Python'}
                # 设置代理 IP
                # "proxy": "http://101.200.153.236:8123",
            }
            yield SplashRequest(url, self.parse, args=args)

    def parse(self, response):
        # 将响应内容生成 Selector，用于数据清洗
        sel = Selector(response)
        # 定义 DongmanItem 对象
        item = DongmanItem()
        info list=sel.xpath('//div[@class="vd-list-cnt"]/ul/li')
        for i in info list:
            item['name'] = ''.join(i.xpath('.//div/div[2]/
```

```
                        a/text()').extract()).strip()
            item['desc'] = ''.join(i.xpath('.//div/div[2]/
                        div[1]/text()').extract()).strip()
            item['viewNumber']=''.join(i.xpath('.//div/div[2]/div[2]/
                        span[1]/span/text()').extract()).strip()
            item['captionsNum']=''.join(i.xpath('.//div/div[2]/div[2]/
                        span[2]/span/text()').extract()).strip()
            yield item
```

在上述 Spider 程序里，动画信息的 URL 地址以类属性 start_urls 表示，并将末端的数字设为动态变量，这样可以得到不同页数的动画信息。Spider 程序信息还重写 start_requests()和 parse()方法，两者实现的功能说明如下：

- start_requests()执行两次循环，每次循环用于构建不同页数的 URL 地址，本例只构建了第一页和第二页的动画信息。
- 每次循环设有变量 headers 和 args，变量 headers 是请求头，args 是设置 scrapy_splash 的请求信息，比如 wait 是等待加载时间、headers 是设置请求头、cookies 是为当前请求添加 Cookies 信息等。
- 最后生成请求对象是由 scrapy_splash 的 SplashRequest()实现，这是在 Scrapy 的 Request()基础上进行自定义，使当前请求在遵循 Scrapy 的规则下实现 Splash 访问 URL 地址。
- parse()的参数 response 是由 scrapy_splash 定义的中间件生成，数据的清洗方式是从开发者工具的 Elements 标签分析网页内容得知。

运行项目之前，还需要开启 Splash 服务器，在电脑上找到 Docker Quickstart Terminal 图标并双击运行。成功启动 Docker 之后，输入 Splash 启动指令 docker run -p 8050:8050 scrapinghub/splash 并按回车键即可，如图 23-17 所示。

```
000@DESKTOP-GOD9Q18 MINGW64 /c/Program Files/Docker Toolbox
$ docker run -p 8050:8050 scrapinghub/splash
2018-11-12 08:18:56+0000 [-] Log opened.
2018-11-12 08:18:56.742827 [-] Splash version: 3.2
2018-11-12 08:18:56.774236 [-] Qt 5.9.1, PyQt 5.9, WebKit 602.1, sip 4.19.
2018-11-12 08:18:56.774657 [-] Python 3.5.2 (default, Nov 23 2017, 16:37:(
2018-11-12 08:18:56.775068 [-] Open files limit: 1048576
2018-11-12 08:18:56.775331 [-] Can't bump open files limit
2018-11-12 08:18:56.913411 [-] Xvfb is started: ['Xvfb', ':167948583', '-s
]
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
2018-11-12 08:18:58.785750 [-] proxy profiles support is enabled, proxy pr
2018-11-12 08:18:59.374965 [-] verbosity=1
2018-11-12 08:18:59.375388 [-] slots=50
2018-11-12 08:18:59.375657 [-] argument_cache_max_entries=500
2018-11-12 08:18:59.376684 [-] Web UI: enabled, Lua: enabled (sandbox: ena
2018-11-12 08:18:59.377172 [-] Server listening on 0.0.0.0:8050
2018-11-12 08:18:59.378630 [-] Site starting on 8050
2018-11-12 08:18:59.379114 [-] Starting factory <twisted.web.server.Site c
```

图 23-17　开启 Splash 服务器

最后在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl finish_opera，启动并运行项目 dongman。项目运行完成后，打开 spiderdb 数据库查看 dongman_db 数据表的数据信息，如图 23-18 所示。

对象 | dongman_db @spiderdb (...

开始事务 | 文本 | 筛选 | 排序 | 导入 | 导出

id	name	desc
1	【360P/DVDrip】网球王子二人的武士+迹部的礼物【空间字幕组	空间字幕组 买了国语版权了
2	【408P/日语/剧场版】网球王子 二人的武士【红旅动漫】	红旅动漫青学网球部受一位
3	【DVDRip】网球王子PRINCE OF TENNIS Movie1【txxz】	网络《二人的武士》在结束
4	【剧场版网球王子】迹部的礼物	迹部为了庆祝桦地的生日，
5	【剧场版网球王子】二人的武士	青学网球部受一位富豪（是
6	【480P/日语生肉】甜蜜小天使/神秘的小寇 第二季 全64+剧场2	网络甜蜜小天使/神秘的小寇

SELECT * FROM `spiderdb`.`dongman_db` LIMIT | 第 1 条记录（共 40 条）于第 1 页

图 23-18　动画信息

23.5　实战：Scrapy+Redis 分布式爬取猫眼排行榜

分布式爬虫好比盖房子一样，如果 1 个人盖一栋房子，他完成任务的时间相对较长，若 20 个人同时盖一栋房子，这样的工作效率就大大提升，完成任务的工作时间也相应减少。分布式爬虫就是将一个爬虫任务分给多个相同的爬虫程序同时执行，而且每个爬虫程序所爬取的内容各不相同。

23.5.1　Scrapy_Redis 实现原理

用 Scrapy 爬虫框架实现分布式爬虫需要借助 Redis 数据库才能实现，它的实现过程与第 18 章爬取 QQ 音乐的实现有所不同。Redis 数据库在此担任了任务队列的作用，负责调度各个爬虫程序的爬取内容，以便有效控制每个爬虫程序之间的重复爬取问题。分布式原理图如图 23-19 所示。

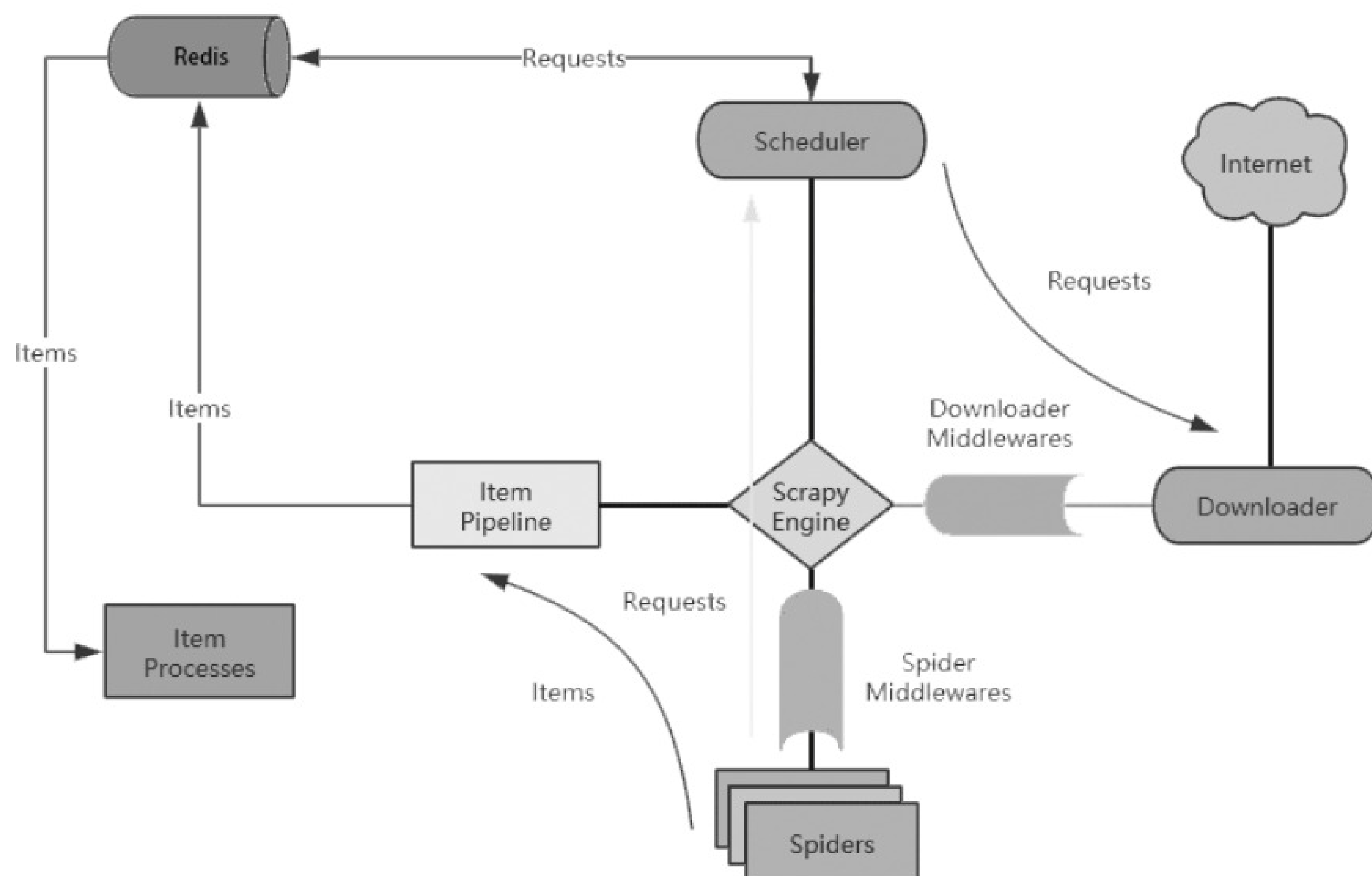


图 23-19　Scrapy_Redis 原理图

Scrapy_Redis 的分布式架构是在 Scrapy 的架构上进行修改和扩展而来，既保留了 Scrapy 的 Twisted 异步框架功能，又新增了分布式功能。新增的分布式功能说明如下：

- Scrapy_Redis 的分布式原理是指多个相同的爬虫程序同时执行，每个爬虫程序在执行期间都会向 Redis 数据库获取任务，根据任务爬取相应的数据。
- 如果爬取数据的 URL 地址已保存在 Redis 数据库，那么爬虫程序不再对该 URL 地址进行爬取。
- 如果 URL 地址不在 Redis 数据库，爬虫程序就会对该 URL 进行数据爬取，并且将 URL 信息写入 Redis 数据库，防止其他爬虫程序重复爬取。

Scrapy_Redis 是 Scrapy 框架的一个扩展功能，它以第三方模块的形式表示，爬虫开发者只需安装该模块即可使用。在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入 pip 安装指令 pip install scrapy-redis，并等待安装完成。然后在 Python 安装目录下查看 scrapy_Redis（Lib\site-packages\scrapy_redis）源码文件，如图 23-20 所示。

软件 (D:) › Python › Lib › site-packages › scrapy_redis ›

名称	修改日期	类型
__pycache__	2018/10/10 14:37	文件夹
__init__.py	2018/10/10 14:37	Python File
connection.py	2018/10/10 14:37	Python File
defaults.py	2018/10/10 14:37	Python File
dupefilter.py	2018/10/10 14:37	Python File
picklecompat.py	2018/10/10 14:37	Python File
pipelines.py	2018/10/10 14:37	Python File
queue.py	2018/10/10 14:37	Python File
scheduler.py	2018/10/10 14:37	Python File
spiders.py	2018/10/10 14:37	Python File
utils.py	2018/10/10 14:37	Python File

图 23-20　Scrapy_Redis 源码文件

Scrapy_Redis 共有 10 个源码文件，每个文件负责实现不同的功能，文件之间存在相互调用的关系，从而实现整个分布式功能。每个文件的功能说明如下：

- __init__.py 是初始化文件，在 Scrapy 调用 Scrapy_Redis 的时候，首先执行 Redis 数据库连接。数据库连接的函数在 connection.py 里定义。
- connection.py 是通过 Redis 模块实现 Redis 数据库连接。函数 get_redis_from_settings()是从 Scrapy 的配置文件 settings.py 读取配置信息，再调用函数 get_redis()实现数据库连接。
- defaults.py 设置 Scrapy-Redis 的基本配置。如果在 Scrapy 项目没有配置相关属性，则使用该文件的配置信息。
- dupefilter.py 重写了 Scrapy 原有的判断重复爬取功能。RFPDupeFilter 是继承

BaseDupeFilter，父类是 Scrapy 原有的判重功能类，RFPDupeFilter 类是在此基础上与 Redis 数据库结合，将已爬取的 URL 地址按一定的规则写入 Redis 数据库。

- picklecompat.py 定义了 loads()和 dumps()函数，这是将数据转化成序列化格式，再将其存储在 Redis 数据库，这样可以解决 Redis 数据库写入数据的格式问题。
- pipelines.py 定义一个 item pipieline 类，它与 Scrapy 的 item pipeline 类是同一个对象，并且调用了 connection.py 的函数，实现 Redis 数据库连接和数据入库。
- queue.py 定义分布式爬虫的任务队列，任务队列的方式有队列、栈和优先级队列，主要为调度器提供调度方式。
- scheduler.py 定义分布式爬虫的调度器，使其代替 Scrapy 原有的调度器，调度方式由 queue.py 的函数实现。在分布式爬虫运行的时候，多台计算机运行同一个 Scrapy 项目，每台计算机的调度器都是相同的，并且连接同一个 Redis 数据库，当一个调度器发生变化的时候，其他也随之变化，因此可以将多个调度器看成一个调度池，每台计算机的爬虫程序由调度池统一管理，从而实现分布式爬虫之间的统一调度。
- spiders.py 重写 Scrapy 原有的 Spider 爬取方式，通过 connection.py 的函数将自定义 Spider 连接到 Redis 数据库，然后由 next_requests()函数从 Redis 中取出 URL 地址进行爬取，Spider 从 Redis 中去除 URL 地址须经过调度器统一调度才能执行。
- utils.py 定义 Scrapy_Redis 的编码格式，使其兼容 Python2 和 Python3 版本。

总的来说，Scrapy_Redis 通过重写 Scrapy 原有的调度器 Scheduler 和 Spider，使 Scheduler、Spider 和 Redis 数据库实现相互连接，并以 Redis 数据库的数据为准，由 Scheduler 统一调度各个 Spider 执行数据爬取，调度器的调度方式和判重功能分别由 queue.py 和 dupefilter.py 实现。

23.5.2 安装 Redis 数据库

在开发 Scrapy 分布式爬虫之前，首先需要安装 Redis 数据库，在 Windows 中安装 Redis 数据库有两种方式：在官网下载压缩包安装或者在 GitHub 下载 MSI 安装程序。前者的数据库版本是最新的，但需要通过指令安装并设置相关的环境配置；后者是旧版本，但安装方法是傻瓜式安装，启动程序后单击按钮即可完成安装。两者的下载地址如下：

```
# 官网下载地址
https://redis.io/download
# github 下载地址
https://github.com/MicrosoftArchive/redis/releases
```

Redis 数据库的安装过程本书就不详细讲述了，读者可以自行查阅相关的资料。除了安装 Redis 数据库之外，还可以安装 Redis 数据库的可视化工具，可视化工具可以帮助初次接触 Redis 的读者了解数据库结构。本书使用 Redis Desktop Manager 作为 Redis 的可视化工具，如图 23-21 所示。

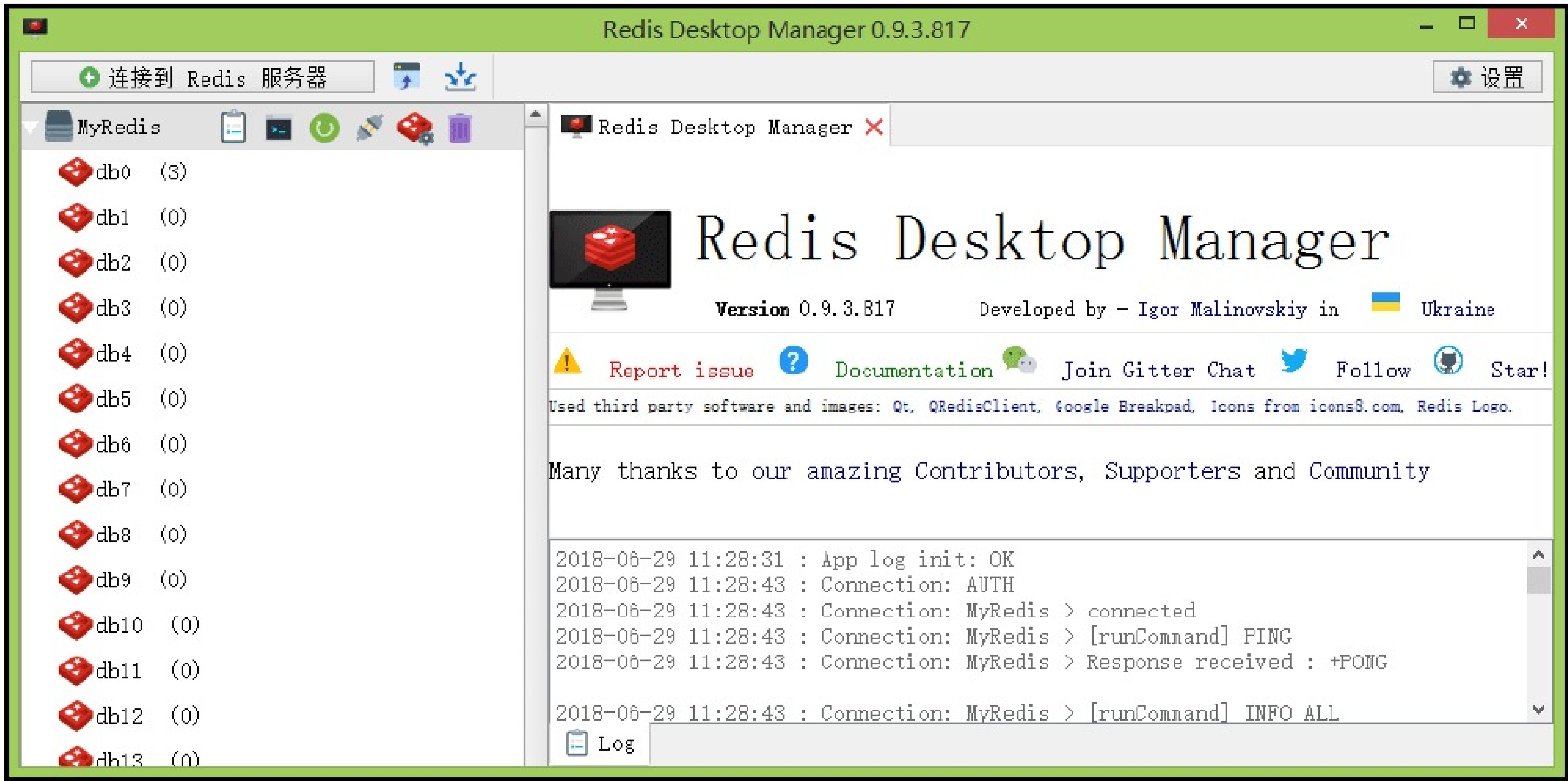


图 23-21 Redis Desktop Manager

23.5.3 网站分析

在浏览器上打开猫眼电影 TOP 100 榜（http://maoyan.com/board/4），并利用开发者工具分析网页内容。由于 Scrapy_Redis 是采用网站 API 的方式爬取目标数据，因此在开发者工具的 Network 标签下分析网站，如图 23-22 所示。

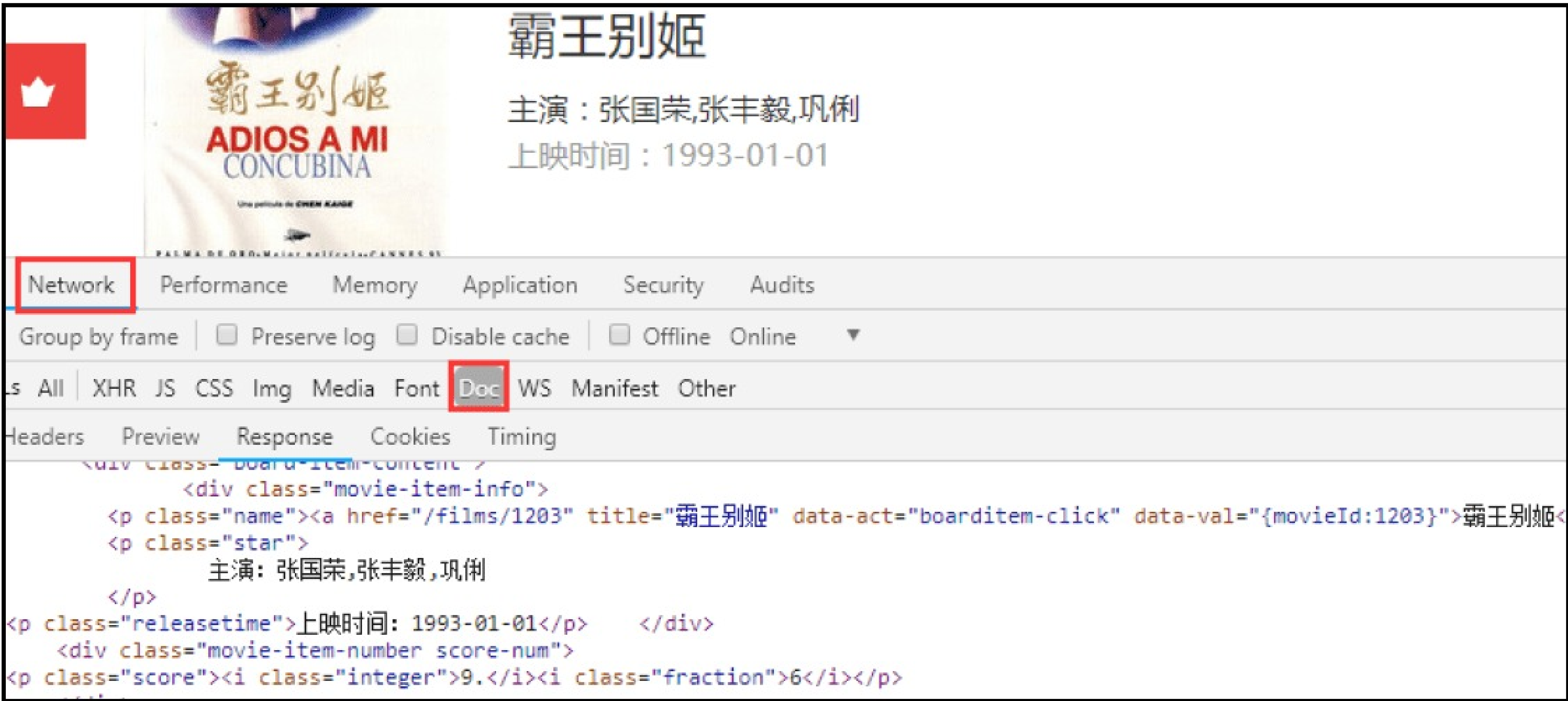


图 23-22 网页分析

从网页结构分析得知，当前网页所有的电影信息在属性 class 为 board-wrapper 的 dl 标签里，每部电影的详细信息在属性 class 为 board-item-content 的 div 标签里。当单击网页最下方的分页按钮的时候，发现 URL 地址随之发生变化，新增了请求参数 offset，这是代表 100 部电影的偏移量，如图 23-23 所示。



图 23-23　URL 地址变化规律

请求参数 offset 从 0 开始，并以 10 进行递增，第一页的参数值为 0、第二页的参数值为 10、第三页的参数值为 20……以此类推，得出请求参数 offset 的变化规律为 p×10，p 代表页数并且从 0 开始计算。

23.5.4　项目设计与实现

根据分析结果设计 Scrapy 项目，在 CMD 命令提示符窗口下创建 Scrapy 项目，项目名为 maoyan，并且在 spiders 文件夹里创建 movieTop.py 文件，整个项目的目录结构如图 23-24 所示。

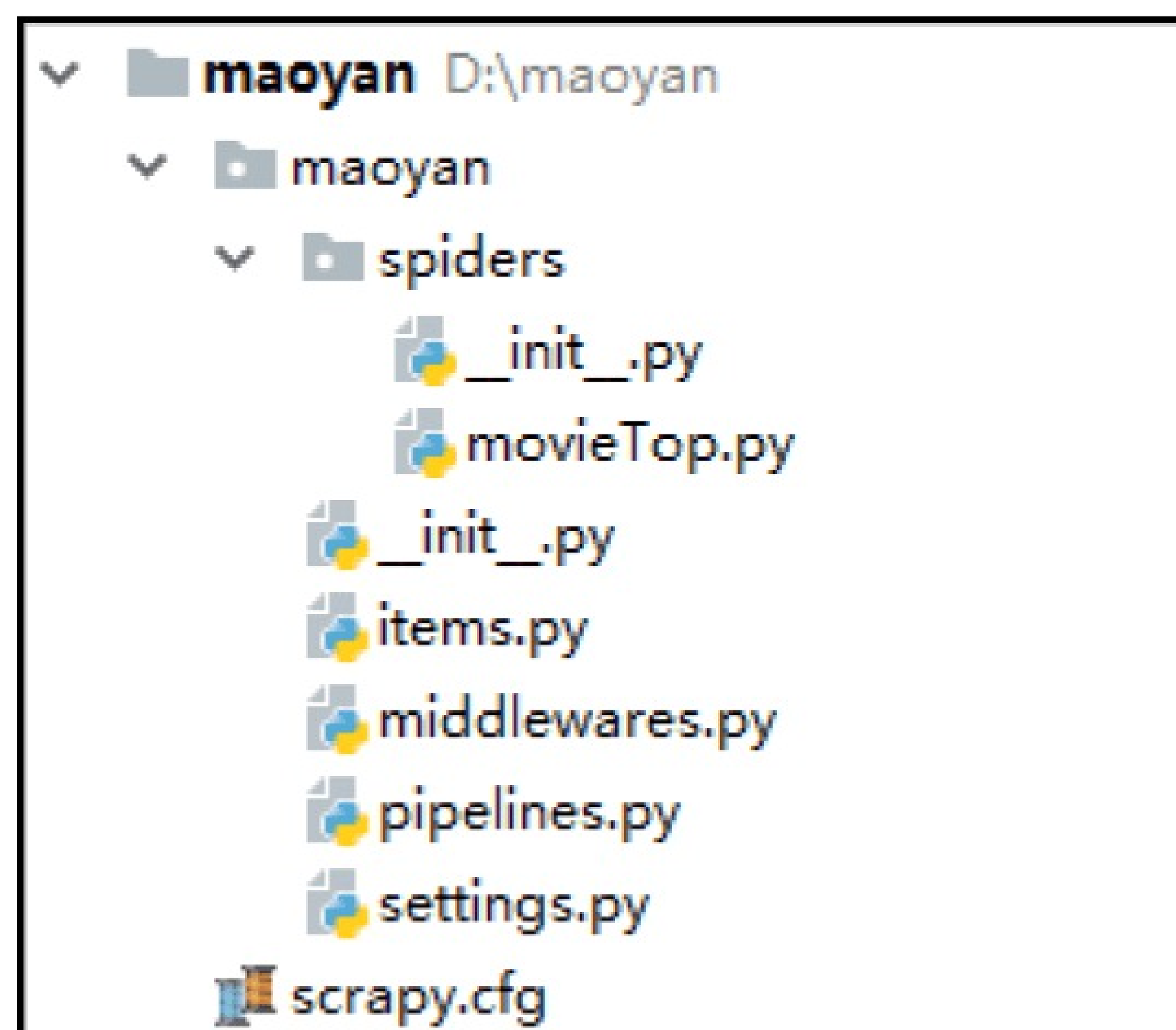


图 23-24　maoyan 的目录结构

在本项目需要加入 Scrapy_Redis 功能模块，在配置文件 settings.py 设置 Scrapy_Redis 的配置属性即可；从网页上爬取的电影名、演员信息、上映时间和评分存储在 MySQL 数据库里，存储过程由 SQLAlchemy 完成。

首先打开配置文件 settings.py，将 Scrapy_Redis 的功能以及相关配置写入文件，代码如下：

```
BOT_NAME = 'maoyan'
```

```
SPIDER MODULES = ['maoyan.spiders']
NEWSPIDER MODULE = 'maoyan.spiders'
# 改为 False
ROBOTSTXT_OBEY = False
# 设置请求头
DEFAULT REQUEST HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,
             application/xml;q=0.9,*/*;q=0.8',
  'Accept-Language': 'en',
  'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                 AppleWebKit/537.36 (KHTML, like Gecko)
                 Chrome/69.0.3497.100 Safari/537.36'
}

# 设置 Scrapy-Redis 的管道 RedisPipeline
ITEM PIPELINES = {
   'maoyan.pipelines.MaoyanPipeline': 300,
   'scrapy redis.pipelines.RedisPipeline': 400
}

# 启用 Redis 调度存储请求队列
SCHEDULER="scrapy redis.scheduler.Scheduler"
# 确保所有的爬虫通过 Redis 去重
DUPEFILTER CLASS="scrapy redis.dupefilter.RFPDupeFilter"
# 允许暂停，Redis 数据不会丢失
SCHEDULER PERSIST = True
# 默认的请求队列顺序
SCHEDULER QUEUE CLASS="scrapy redis.queue.SpiderPriorityQueue"
# # 队列形式，请求先进先出
# SCHEDULER QUEUE CLASS="scrapy redis.queue.SpiderQueue"
# # 栈形式，请求先进后出
# SCHEDULER QUEUE CLASS="scrapy redis.queue.SpiderStack"
# 设置 Redis 数据库连接信息
REDIS URL = 'redis://localhost:6379/'
# MySQL 数据库连接信息
MYSQL CONNECTION = 'mysql+pymysql://root:1234@
                    localhost/spiderdb?charset=utf8mb4'
```

配置文件 settings.py 主要配置 Scrapy_Redis 的功能，其中配置属性 SCHEDULER 是改变 Scrapy 原有的调度器。当项目运行的时候，Scrapy 从配置文件读取配置信息，根据配置信息运行 Scrapy_Redis 的功能，使得整个项目的调度器 Scheduler 和 Spider 皆由 Scrapy_Redis 定义，从而实现分布式爬虫。

接着打开项目的 items.py 文件，将项目需要存储的字段在此文件进行定义。每部电影的电影名、演员信息、上映时间和评分都与字段 name、desc、viewNumber 和 captionsNum 相互对应，如下所示：

```
import scrapy
class MaoyanItem(scrapy.Item):
    movieName = scrapy.Field()
    performer = scrapy.Field()
    releasetime = scrapy.Field()
    score = scrapy.Field()
```

```
        pass
```

最后在项目的 pipelines.py 文件编写相关的数据存储过程。该文件里定义了两个类，分别代表数据表映射类 scrapy_db 和数据存储 MaoyanPipeline，定义方式如下：

```
from sqlalchemy import *
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative base
from scrapy.conf import settings

# 定义映射类
Base = declarative base()
class scrapy db(Base):
      tablename   = 'maoyan db'
    id = Column(Integer(), primary_key=True)
    movieName = Column(String(100))
    performer = Column(String(100))
    releasetime = Column(String(200))
    score = Column(String(100))

class MaoyanPipeline(object):
    def   init  (self):
        # 初始化，连接数据库
        conntion = settings['MYSQL CONNECTION']
        engine=create engine(conntion,echo=False,pool size=2000)
        DBSession = sessionmaker(bind=engine)
        self.SQLsession = DBSession()
        # 创建数据表
        Base.metadata.create all(engine)

    def process_item(self, item, spider):
        # 入库处理
        self.SQLsession.execute(scrapy db.  table  .insert(),
                               {'movieName': item['movieName'],
                                'performer': item['performer'],
                                'releasetime': item['releasetime'],
                                'score': item['score']})
        self.SQLsession.commit()
        return item
```

23.5.5　开发 Spider 程序

由于网页的 URL 地址只有请求参数 offset，参数值的变化规律为 p×10，p 代表页数并且从 0 开始计算。spider 程序只需循环 10 次，由每次循环的次数来构建 URL 地址即可获取不同页数的电影信息，实现代码如下：

```
from maoyan.items import MaoyanItem
from scrapy.selector import Selector
from scrapy.spider import Spider, Request
class MovieSpider(Spider):
    # 属性 name 必须设置，而且是唯一命名，用于运行爬虫
    name = "Movie"
```

```
    # 设置允许访问域名
    allowed domains = ["maoyan.com"]
    # 设置 URL
    start urls='http://maoyan.com/board/4?offset=%s'
    # 重写 start_requests
    def start requests(self):
        # TOP100 的电影共 10 页，则循环 10 次
        for page in range(10):
            url = self.start urls %(str(page * 10))
            yield Request(url=url,callback=self.parse)

    def parse(self, response):
        # 将响应内容生成 Selector，用于数据清洗
        sel = Selector(response)
        # 定义 DoubanItem 对象
        item = MaoyanItem()
        infoList = sel.xpath('//dl[@class="board-wrapper"]//dd')
        for c in infoList:
            item['movieName'] = ''.join(c.xpath('.//p[@class=
                              "name"]//text()').extract()).strip()
            item['performer'] = ''.join(c.xpath('.//p[@class=
                              "star"]//text()').extract()).strip()
            item['releasetime'] = ''.join(c.xpath('.//p[@class=
                             "releasetime"]//text()').extract()).strip()
            item['score'] = ''.join(c.xpath('.//p[@class=
                          "score"]//text()').extract()).strip()
            yield item
```

上述代码的类属性 start_urls 为猫眼电影 TOP 100 排行榜的 URL 地址，请求参数 offset 设为字符串格式化，这样可动态设置参数 offset 的参数值。spider 程序定义了 start_requests()和 parse()方法，说明如下：

- start_requests()方法实现 10 次循环，将当前循环的次数乘以 10 即可得出请求参数 offset 的数值，从而构建不同页面的 URL 地址。
- parse()方法是将网页的响应内容进行数据清洗，清洗后的数据传递给 MaoyanPipeline()实现数据入库。

在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl Movie，启动并运行项目 maoyan。项目运行完成后，打开 spiderdb 数据库查看 maoyan_db 数据表的数据信息，如图 23-25 所示。

同时打开 Redis 可视化工具 Redis Desktop Manager，查看 Redis 数据库的数据信息，发现 Redis 数据库分别存储了 Scrapy_Redis 的判重数据 dupefilter 和爬取数据 items。判重数据 dupefilter 是将网页的 URL 地址按一定的规则写入 Redis 数据库；爬取数据 items 是将网页的响应内容清洗后写入 Redis 数据库，如图 23-26 所示。

id	movieName	performer	releasetime	score
1	神偷奶爸	主演：史蒂夫·卡瑞尔,杰森·席格尔,拉塞尔·布兰德	上映时间：2010-(	9.0
2	飞屋环游记	主演：爱德华·阿斯纳,乔丹·长井,鲍勃·彼德森	上映时间：2009-(	8.9
3	鬼子来了	主演：姜文,姜宏波,陈强	上映时间：2000-(	8.9
4	魔女宅急便	主演：高山南,佐久间玲,户田惠子	上映时间：1989-(	8.8

SELECT * FROM `spiderdb`.`maoyan_db` l　第 1 条记录（共 100 条）于第 1 页

图 23-25　电影排行榜信息

MyDb::db0::Movie:dupefilter

SET: Movie:dupefilter

row	value
1	92f27e7b845f3a8f···
2	d55df7f2a944b590···
3	54e639d6cd6526f2···

图 23-26　Redis 数据库的数据信息

由于 Redis 数据库已存储相关数据，当再次执行项目 maoyan 的时候，如果网站内容没有更新，项目 maoyan 不会再重复爬取。因为 Redis 数据库已有相同的数据，这样可以防止分布式的爬虫程序重复爬取，保证了数据的唯一性。

23.6　分布式爬虫与增量式爬虫

从 Scrapy_Redis 的分布式爬虫原理得知，当前请求的 URL 地址和响应内容都已保存在 Redis 数据库的时候，Scrapy 不再对当前请求发送 HTTP 请求，而是直接执行下一个请求，这样可以防止分布式的爬虫程序重复爬取，保证了数据的唯一性。

根据分布式爬虫原理可以衍生出一个新的使用方法——增量式爬虫。增量式爬虫是在已保存网站部分数据的情况下，当再次运行爬虫的时候，爬虫对已有的数据不再重复爬取，只爬取数据库尚未保存的数据。分布式爬虫 Scrapy_Redis 也可以作为增量式爬虫，此外还可以在 Scrapy 项目里自主开发增量式爬虫，实现原理与 Scrapy_Redis 的大同小异。

自主开发增量式爬虫的方式有两种：基于管道实现增量式和基于中间件实现增量式。两者在 Scrapy 的不同文件里实现，说明如下：

- 基于管道实现增量式爬虫是在 pipelines.py 文件的 process_item()方法里判断数据是否已保存在 Redis 数据库，如果存在则不做数据入库处理，反之将当前数据写入 Redis 数据库和目标数据库。
- 基于中间件实现增量式爬虫是在 middlewares.py 文件里定义中间件，判断当前请求的 URL 地址是否已在 Redis 数据库，如果存在则跳过当前请求并直接执行下一个请求，反之将当

前的 URL 地址写入 Redis 数据库并对该请求往下执行。

23.6.1 基于管道实现增量式

基于管道实现增量式是将 Scrapy_Redis 的 pipelines.py 单独使用，它最大的优点是对已访问的 URL 地址重复访问并获取数据，这样可及时更新数据的动态变化，常用于排行榜、论坛贴吧等网站的爬取；但这也是一个最大的缺点，因为它会对 URL 地址重复访问，如果网站数据固定不变就会造成网络资源浪费，同时也增大了反爬虫机制检测的风险。

实现管道增量式爬虫可以使用 Scrapy_Redis 的 pipelines.py 文件的 RedisPipeline 类，不过它会涉及到 Scrapy_Redis 其他文件的使用。为了简化功能的复杂度，可以根据原理在项目的 pipelines.py 文件编写相应功能即可，本节以豆瓣电影评论为例，讲述如何在 Scrapy 项目实现管道增量式爬虫开发。

本节项目来自第 23.3 节的 douban 项目，将项目的中间件 Selenium 去掉，这是为了简化项目功能，方便读者理解，由于功能发生改变，各个文件代码也进行相应的调整。首先打开配置文件 settings.py，删除中间件 Selenium 的相关配置，代码如下：

```
BOT_NAME = 'douban'
SPIDER_MODULES = ['douban.spiders']
NEWSPIDER_MODULE = 'douban.spiders'
ROBOTSTXT_OBEY = False

DEFAULT_REQUEST_HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,
             application/xml;q=0.9,*/*;q=0.8',
  'Accept-Language': 'en',
  # 加上 User-Agent，否则提示 403 错误信息
  'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                 AppleWebKit/537.36 (KHTML, like Gecko)
                 Chrome/69.0.3497.100 Safari/537.36'
}

DOWNLOADER_MIDDLEWARES = {
   'douban.middlewares.DoubanDownloaderMiddleware': 543,
}

ITEM_PIPELINES = {
   'douban.pipelines.DoubanPipeline': 300,
}

import os
BASE_DIR = os.path.dirname(os.path.realpath(__file__))
CONF = os.path.join(BASE_DIR, 'conf.ini')
MYSQL_CONNECTION = 'mysql+pymysql://root:1234@
                    localhost/spiderdb?charset=utf8mb4'
```

打开项目的 items.py 文件，将项目存储的字段定义为 movieInfo，该字段将以列表格式表示，列表的每个元素以字典表示，字典里包含了电影 ID 和评论内容，代码如下：

```
import scrapy
```

```
class DoubanItem(scrapy.Item):
    movieInfo = scrapy.Field()
```

在项目的 pipelines.py 文件中修改数据存储过程，将数据存储 DongmanPipeline 重新定义，分别对初始化方法__init__()和 process_item()进行重写，代码如下：

```
from sqlalchemy import *
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative base
from scrapy.conf import settings
import redis

# 定义映射类
Base = declarative base()
class scrapy db(Base):
      tablename    = 'douban db'
    id = Column(Integer(), primary key=True)
    movieId = Column(String(100))
    comment = Column(String(2000))

class DoubanPipeline(object):
    def   init   (self):
        # 连接 Redis 数据库
        self.redis db = redis.Redis(host='127.0.0.1',port=6379,db=1)
        self.redis data dict = 'keys'
        # 初始化，连接数据库
        conntion = settings['MYSQL CONNECTION']
        engine = create engine(conntion,echo=False,pool size=2000)
        DBSession = sessionmaker(bind=engine)
        self.SQLsession = DBSession()
        # 创建数据表
        Base.metadata.create all(engine)

    def process item(self, item, spider):
        # 判断数据库 Redis 是否存在 URL
        for i, v in enumerate(item['movieInfo']):
            if self.redis db.hexists(self.redis data dict, v):
                # 若存在，输出数据库已存在的数据
                print('数据库已经存在该条数据')
            else:
                # 不存在，写入数据库 Redis
                self.redis db.hset(self.redis data dict, v, 0)
                # 入库处理
                self.SQLsession.execute(scrapy db.  table  .insert(),
                                    {'comment': v['comment'].replace("\n", ""),
                                     'movieId': v['movieId']})
                self.SQLsession.commit()
        return item
```

数据存储 DoubanPipeline 通过重写初始化方法__init__()，在 DoubanPipeline()实例化的时候，使用 redis 模块连接 Redis 数据库以及读取配置文件 settings.py 的数据库连接信息，由 SQLAlchemy 实现 MySQL 数据库连接。

方法 process_item()是遍历 Scrapy 引擎传递的参数 item，每次遍历代表某部电影的某条评论，

将每条评论在 Redis 数据库中进行查找，若存在，则提示数据库已经存在该条数据，否则对该数据分别写入 Redis 和 MySQL 数据库。

最后打开项目 spiders 文件夹的 movie.py 文件，将 spider 程序进行调整，由于已去掉中间件 Selenium 和改变了数据存储方式，所以分别对 spider 程序的 start_requests()和 parse()进行修改，代码如下：

```
from douban.items import DoubanItem
from scrapy.selector import Selector
from scrapy.spider import Spider, Request
import configparser

class MovieSpider(Spider):
    # 属性 name 必须设置，而且是唯一命名，用于运行爬虫
    name = "Movie"
    # 设置允许访问域名
    allowed domains = ["https://movie.douban.com"]
    # 设置 URL
    start urls = 'https://movie.douban.com/subject/%s/comments?
                  start=%s&limit=20&sort=new score&status=P'
    # 重写 start_requests
    def start requests(self):
        # 读取配置文件，获取电影 ID 并生成列表
        conf = configparser.ConfigParser()
        urlsList = []
        conf.read(self.settings.get('CONF'))
        temp = conf['config']
        if 'movieId' in temp.keys():
            urlsList = conf['config']['movieId'].split(',')
        for u in urlsList:
            # 每部电影爬取两页的评论
            for page in range(2):
                url = self.start urls %(str(u), str(page * 20))
                yield Request(url=url,meta={'movieId': str(u)},
                              callback=self.parse)

    def parse(self, response):
        # 将响应内容生成 Selector，用于数据清洗
        sel = Selector(response)
        # 定义 DoubanItem 对象
        item = DoubanItem()
        comments=sel.xpath('//div[@id="comments"]//div[@class="comment"]')
        commentsList = []
        for c in comments:
            movieId = response.meta['movieId']
            comment=''.join(c.xpath('.//p//span//text()').extract()).strip()
            commentsList.append(dict(movieId=movieId,comment=comment))
        item['movieInfo'] = commentsList
        yield item
```

在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl Movie，启动并运行项目 douban。项目运行完成后，打开 MySQL 数据库的 douban_db 数据表和 Redis 数据库，分别查看数据信息，如图 23-27 所示。



图 23-27　MySQL 和 Redis 的数据信息

当重复运行项目 douban 的时候，程序依然能向网站发送 HTTP 请求，并且将爬取的数据进行清洗处理，直到在数据存储的时候才会提示“数据库已经存在该条数据”。

23.6.2　基于中间件实现增量式

基于中间件实现增量式爬虫是在发送 HTTP 请求之前，首先对该请求的 URL 地址进行判断，如果该 URL 地址在此之前已发送过 HTTP 请求，则本次请求不再往下执行，这样可以避免同一个 URL 地址重复访问。因此，自定义中间件主要对当前请求的 URL 地址进行判断，根据不同的判断结果执行不同的处理方式。

本节同样以豆瓣电影评论为例，讲述如何在 Scrapy 项目实现中间件增量式爬虫开发。将第 23.3 节的 douban 项目的中间件 Selenium 去掉，并对各个文件代码也进行相应的调整。首先打开配置文件 settings.py，删除中间件 Selenium 的相关配置以及注册激活中间件 RedisMiddleware，代码如下：

```
BOT_NAME = 'douban'
SPIDER_MODULES = ['douban.spiders']
NEWSPIDER_MODULE = 'douban.spiders'
ROBOTSTXT_OBEY = False

DEFAULT_REQUEST_HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,
             application/xml;q=0.9,*/*;q=0.8',
  'Accept-Language': 'en',
  # 加上 User-Agent，否则提示 403 错误信息
  'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                 AppleWebKit/537.36 (KHTML, like Gecko)
                 Chrome/69.0.3497.100 Safari/537.36'
}
DOWNLOADER_MIDDLEWARES = {
   'douban.middlewares.DoubanDownloaderMiddleware': 543,
```

```
    'douban.middlewares.RedisMiddleware': 300,
}
ITEM PIPELINES = {
    'douban.pipelines.DoubanPipeline': 300,
}
import os
BASE_DIR = os.path.dirname(os.path.realpath(__file__))
CONF = os.path.join(BASE DIR, 'conf.ini')
MYSQL CONNECTION = 'mysql+pymysql://root:1234@
                     localhost/spiderdb?charset=utf8mb4'
```

项目的 items.py 和 pipelines.py 文件无需修改，沿用项目原有的实现方式即可。然后将 spider 程序的功能代码进行调整，去除参数 meta 的 usedSelenium 属性，修改代码如下：

```
from douban.items import DoubanItem
from scrapy.selector import Selector
from scrapy.spider import Spider, Request
import configparser

class MovieSpider(Spider):
    # 属性 name 必须设置，而且是唯一命名，用于运行爬虫
    name = "Movie"
    # 设置允许访问域名
    allowed domains = ["https://movie.douban.com"]

    # 设置 URL
    start urls = 'https://movie.douban.com/subject/%s/comments?
                 start=%s&limit=20&sort=new score&status=P'
    # 重写 start_requests
    def start requests(self):
        # 读取配置文件，获取电影 ID 并生成列表
        conf = configparser.ConfigParser()
        urlsList = []
        conf.read(self.settings.get('CONF'))
        temp = conf['config']
        if 'movieId' in temp.keys():
            urlsList=conf['config']['movieId'].split(',')

        for u in urlsList:
            # 每部电影爬取两页的评论
            for page in range(2):
                url = self.start urls %(str(u), str(page * 20))
                yield Request(url=url, meta={'movieId': str(u)},
                              callback=self.parse)

    def parse(self, response):
        # 将响应内容生成 Selector，用于数据清洗
        sel = Selector(response)
        # 定义 DoubanItem 对象
        item = DoubanItem()
        comments=sel.xpath('//div[@id="comments"]//div[@class="comment"]')
        for c in comments:
            item['movieId'] = response.meta['movieId']
            item['comment'] = ''.join(c.xpath('.//p//span//
```

```
                              text()').extract()).strip()
            yield item
```

配置文件 settings.py 已注册激活中间件 RedisMiddleware，因此在 middlewares.py 文件需要定义中间件 RedisMiddleware，该中间件是实现增量式爬虫功能，代码如下：

```
# 自定义 Redis 中间件
from scrapy.http import HtmlResponse
import redis
class RedisMiddleware(object):
    def __init__(self):
        self.redis_db=redis.Redis(host='127.0.0.1',port=6379,db=1)
        self.redis_data_dict = 'keys'

    def process_request(self, request, spider):
        # 判断数据库 Redis 是否存在 URL
        if self.redis_db.hexists(self.redis_data_dict, request.url):
            # 若存在，抛出 500 异常
            return HtmlResponse(url=request.url, status=500, request=request)
        else:
            # 不存在，写入数据库 Redis，并将数据写入 MySQL 数据库
            self.redis_db.hset(self.redis_data_dict, request.url, 0)
            return None
```

自定义中间件 RedisMiddleware 重新定义初始化方法__init__()和 process_request()方法，两者的说明如下：

- 初始化方法__init__()是为 process_request()方法提供 Redis 数据库连接对象，并将对象以类属性 redis_db 表示；类属性 redis_data_dict 是定义 Redis 数据库的 HASH，可理解为数据表的命名。
- process_request()方法是将类属性 redis_data_dict 和参数 request 的 URL 地址传入类属性 redis_db，这样可以判断参数 request 的 URL 地址是否记录在 Redis 数据库
- 如果 URL 地址已记录 Redis 数据库，则说明当前请求在此之前已有访问记录，中间件就会抛出 HTTP 500 异常，这代表终止当前请求的处理。
- 如果 URL 地址尚未记录 Redis 数据库，则说明当前请求尚未生成访问记录，中间件会将 URL 地址写入 Redis 数据库，并且对当前请求执行相应的数据爬取。

在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl Movie，启动并运行项目 douban。项目运行完成后，打开 MySQL 数据库的 douban_db 数据表和 Redis 数据库，分别查看数据信息，如图 23-28 所示。

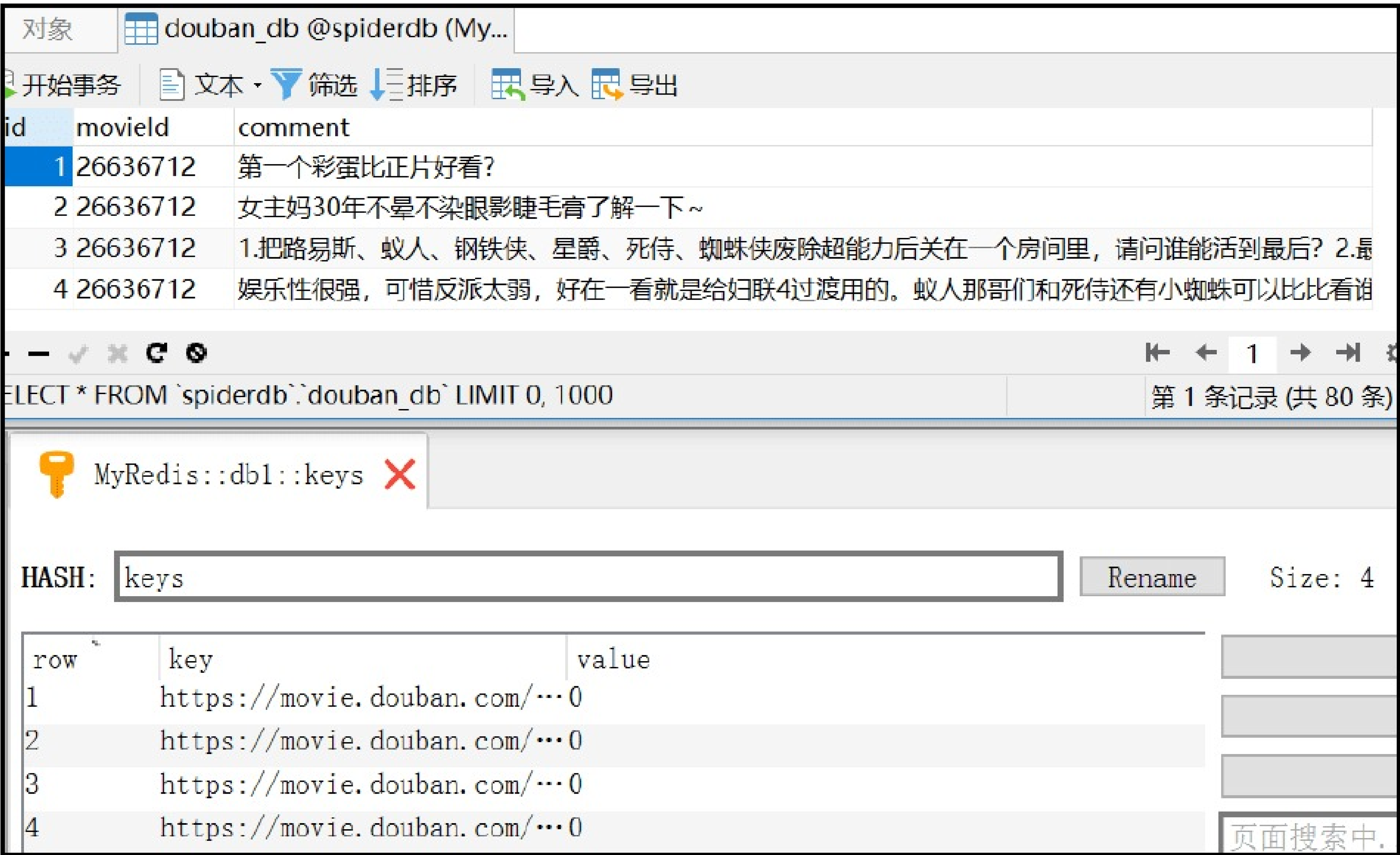


图 23-28　MySQL 和 Redis 的数据信息

当重复运行项目 douban 的时候，项目中所有的 URL 地址已被记录在 Redis 数据库中，因此所有请求都被终止执行，中间件就会抛出 HTTP 500 错误码，如图 23-29 所示。



图 23-29　终止请求

23.7　本 章 小 结

Scrapy 的自定义开发主要体现在 Scrapy 的中间件上，也就是 Scrapy 项目里的 middlewares.py 文件，该文件以类的形式定义中间件，在配置文件 settings.py 注册激活中间件，当项目运行的时候，Scrapy 会自动调用自定义中间件。

创建 Scrapy 项目的时候，middlewares.py 文件默认定义了两个中间件，分别是 SpiderMiddleware 和 DownloaderMiddleware，前者是爬虫中间件，介于 Scrapy 引擎和爬虫之间的框架，主要工作是处理爬虫的响应输入和请求输出；后者是下载器中间件，位于 Scrapy 引擎和下载器之间的框架，处理引擎与下载器之间的请求及响应。两者的工作流程如下：

- Scrapy 向爬取网站发送 HTTP 请求是由中间件 DownloaderMiddleware 的 process_request()函数实现。
- 请求发送成功后，调用 DownloaderMiddleware 的 process_response()函数生成相应的响应内容，并将响应内容发送给 Scrapy 引擎。
- Scrapy 引擎将响应内容传递给 SpiderMiddleware 的 process_spider_input()函数处理，根据开发者编写的 spider 程序来对响应内容进行清洗处理。
- SpiderMiddleware 将处理结果返回给 Scrapy 引擎，这个过程是由 process_spider_output()函数实现。而 Scrapy 引擎再将结果传递给 Item Pipeline，从而实现数据存储。

中间件的自定义是实现类的定义，该类可选择是否继承某个父类，比如定义 MyDownloaderMiddleware 中间件，该中间件可以继承 objects 类（Python 的新式类）；还可以继承 Scrapy 的内置中间件，使得自定义中间件具有内置中间件的功能。

Scrapy 不仅能自定义中间件，还可以将中间件结合其他模块实现不同的爬取方式。在 Scrapy 框架上使用 Selenium 模块实现爬虫开发是常见的手段之一，因为 Selenium 可以模拟用户访问浏览器，从中爬取目标数据，实现过程较为简单，而且能绕开各种反爬虫策略。

Scrapy 框架也可以与 Splash 模块结合使用，从 Scrapy 框架结构可知，使用 Splash 比 Selenium 更有优势，因为 Splash 是一个异步框架，它与 Scrapy 框架能完美结合。

分布式爬虫好比盖房子一样，如果 1 个人盖一栋房子，他完成任务的时间相对较长，若 20 个人同时盖一栋房子，这样的工作效率就会大大提升，完成任务的工作时间也相应减少。分布式爬虫是将一个爬虫任务分给多个相同的爬虫程序同时执行，而且每个爬虫程序所爬取的内容各不相同。

增量式爬虫是在已保存网站部分数据的情况下，当再次运行爬虫的时候，爬虫对已有的数据不再重复爬取，只爬取数据库尚未保存的数据。分布式爬虫 Scrapy_Redis 也可以作为增量式爬虫，此外还可以在 Scrapy 项目里自主开发增量式爬虫，实现原理与 Scrapy_Redis 的大同小异。

自主开发增量式爬虫的方式有两种：基于管道实现增量式和基于中间件实现增量式。两者在 Scrapy 的不同文件里实现，说明如下：

- 基于管道实现增量式爬虫是在 pipelines.py 文件的 process_item()方法里判断数据是否已保存在 Redis 数据库，如果存在则不做数据入库处理，反之将当前数据写入 Redis 数据库和目标数据库。
- 基于中间件实现增量式爬虫是在 middlewares.py 文件里定义中间件，判断当前请求的 URL 地址是否已在 Redis 数据库，如果存在则跳过当前请求并直接执行下一个请求，反之将当前的 URL 地址写入 Redis 数据库并对该请求往下执行。

第 24 章

实战：爬取链家楼盘信息

24.1 项目分析

本章通过实战的形式来深入讲述 Scrapy 的使用方法，我们以链家的二手房信息为爬取对象，分别爬取房屋的信息和所在小区的基本信息。

在浏览器上访问链家二手房的网址（https://gz.lianjia.com/ershoufang/pg1/），发现房屋信息是以列表的形式表示，并且设置为分页功能；每一页有 30 条房屋信息，每个城市只提供 100 页的房屋信息，如图 24-1 所示。



图 24-1　房屋列表

从图 24-1 中得知，每当单击不同页数的时候，房屋列表的 URL 地址会随之变化。比如单击第二页的时候，URL 末端变为 pg2、第三页的 URL 末端变成 pg3……以此类推，URL 末端数字代表当前页数，因此对该数字进行动态设置即可获取不同页数的房屋列表信息。

在每一页的房屋列表里含有房屋详情页的 URL 地址，通过这些 URL 地址可以爬取房屋的基本信息。以某房屋详情页为例，项目需要爬取房屋的 ID、售价和基本信息，如图 24-2 所示。

图 24-2　房屋信息

房屋详细页还可以找到售价和所属小区的 URL 地址，通过该 URL 地址可以进入小区详情页，从而爬取小区的相关信息，如图 24-3 所示。

图 24-3　小区的 URL 地址

从小区的 URL 地址进入小区详情页可以发现，URL 末端的一串数字代表小区 ID，用于标注小区的唯一性；在小区详情页需要爬取小区的名称、位置、建筑年份、物业公司等相关信息，并且还要将小区与房屋之间的数据相互关联，如图 24-4 所示。

https://gz.lianjia.com/xiaoqu/2110343238861283/

建筑年代 1995年建成
建筑类型 塔楼/板楼
物业费用 0.6元/平米/月
物业公司 广州市番禺区美图物业管理有限公司
开发商 番禺区沙湾房地产开发公司
楼栋总数 60栋
房屋总数 1357户
附近门店 暂无门店信息

图 24-4　小区信息

综合上述分析，本项目主要对三个网页进行数据爬取：房屋列表页、房屋详情页和小区详情页，爬取的方向说明如下：

- 根据房屋列表页的 URL 地址构造规律，动态设置 URL 末端的页数来获取全部房屋详情页的 URL 地址。这个获取过程涉及两个循环：页数循环和每页的房屋列表循环；前者是循环 100 页的房屋列表，后者获取每页房屋列表的房屋详情页 URL 地址。
- 房屋详情页的 URL 地址末端的一串数字代表房屋 ID，用来标记房屋的唯一性。在房屋详细页里除了爬取房屋的基本信息之外，还能爬取小区详情页的 URL 地址，从而访问小区详情页爬取小区信息。
- 小区详情页的 URL 地址末端的一串数字代表小区 ID，这是标记小区的唯一性。在小区详情页爬取小区基本信息之外，还要将小区和房屋的数据相互关联，因为会出现一个小区有多套房屋出售的情况。

三个网页所爬取的数据信息都可以在开发者工具 Network 标签下的 Doc 分类标签进行分析，详细的分析过程本书不再讲述，我们将目标数据在 HTML 源码的大概位置以表格的形式加以说明，根据这个位置即可找到每条数据的具体位置，如表 24-1 所示。

表 24-1　目标数据信息

数据	所属页面	HTML 源码位置
全部房屋信息	房屋列表页	属性 class 为 sellListContent 的 ul 标签
每套房屋详情页的 URL 地址	房屋列表页	属性 class 为 title 的 div 标签
房屋售价	房屋详情页	属性 class 为 price 的 div 标签
房屋的基本属性	房屋详情页	属性 class 为 base 的 div 标签
房屋的交易属性	房屋详情页	属性 class 为 transaction 的 div 标签
小区详情页的 URL 地址	房屋详情页	属性 class 为 info 的 a 标签
小区名称	小区详情页	属性 class 为 detailTitle 的 h1 标签
小区位置	小区详情页	属性 class 为 detailDesc 的 div 标签
小区基本信息	小区详情页	属性 class 为 xiaoquInfoContent 的 span 标签

最后从三个网页的域名得知，域名“gz.lianjia.com”的 gz 代表广州，如果切换到其他城市，域名会随之变化，比如深圳的域名为“sz.lianjia.com”或上海的域名为“sh.lianjia.com”。对于这种变化，我们也可以动态设置域名来实现各个城市的爬取。

24.2 创建项目

现在对爬取的网站有了一定的了解，接下来我们创建 Scrapy 爬虫项目。将项目命名为 lianjia，打开 CMD 命令提示符窗口，将当前的路径切换到其他磁盘，然后输入创建指令：

```
scrapy startproject lianjia
```

项目创建完成后，在项目中的 spiders 文件夹里创建 houseSpider.py 文件，该文件用来实现 Spider 功能，用于编写爬虫规则；在配置文件 settings.py 的同一目录下创建 conf.ini 配置文件，conf.ini 文件用于动态设置各个城市的域名信息。

最后在 PyCharm 中打开项目所在的文件夹，目录结构如图 24-5 所示。

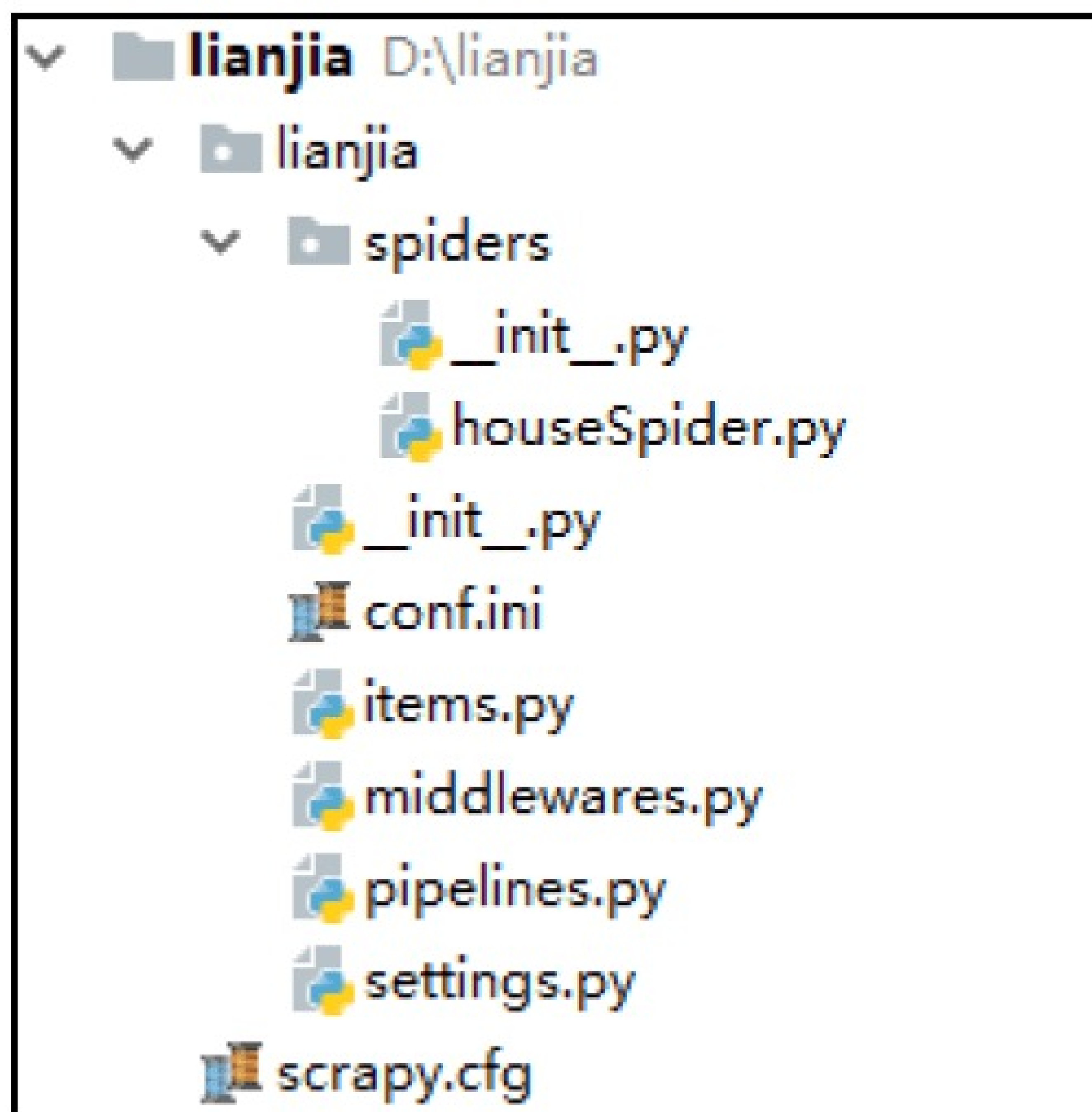


图 24-5　目录结构

24.3 项目配置

从网站分析结果来看，整个项目的开发难度相对较为简单，三个页面的 URL 地址构造规律、响应内容和数据位置都一目了然。因此，项目 lianjia 只需使用 Scrapy 的基本配置即可，配置代码如下：

```
BOT_NAME = 'lianjia'
SPIDER_MODULES = ['lianjia.spiders']
NEWSPIDER_MODULE = 'lianjia.spiders'
# 改为 False
```

```
ROBOTSTXT_OBEY = False
# 设置请求头
DEFAULT_REQUEST_HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,
             application/xml;q=0.9,*/*;q=0.8',
  'Accept-Language': 'zh-CN,zh;q=0.9',
  'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                 AppleWebKit/537.36 (KHTML, like Gecko)
                 Chrome/69.0.3497.100 Safari/537.36',
}
# 数据库连接信息
MYSQL_CONNECTION = 'mysql+pymysql://root:1234@
                    localhost/spiderdb?charset=utf8mb4'
import os
BASE_DIR = os.path.dirname(os.path.realpath(__file__))
CONF = os.path.join(BASE_DIR, 'conf.ini')
# 注册激活管道类
ITEM_PIPELINES = {
   'lianjia.pipelines.LianjiaPipeline': 300,
}
```

从上述代码看出，项目分别对 Item Pipelines、数据库信息、请求头和配置文件 conf.ini 进行配置，各个配置说明如下。

- Item Pipelines：在创建项目时，默认配置了类 LianjiaPipeline，用于实现数据的存储功能，具有的存储功能代码还需要开发者自行编写。
- 数据库信息：该配置属于自定义配置信息，变量 MYSQL_CONNECTION 以字符串格式表示，变量值是 SQLAlchemy 连接数据库语句。数据库系统为本地数据库系统，数据库为 spiderdb。
- 请求头：配置默认的请求头内容，如果项目中发送 HTTP 请求并没有指定请求头，就默认使用该配置作为请求头。
- 配置文件 conf.ini：用于动态设置各个城市的域名信息，由 os 模块读取 settings.py 同目录下的配置文件 conf.ini 的文件路径，并将文件路径设为配置属性 CONF。

除此之外，还可以配置并发数和下载延时等相关信息，读者可根据以下代码自行配置：

```
# Configure maximum concurrent requests performed by Scrapy (default: 16)
# 设置并发数，Scrapy 默认同一时间可并发 16 个请求
#CONCURRENT_REQUESTS = 32
# Configure a delay for requests for the same website (default: 0)
# See
http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay
# See also autothrottle settings and docs
# 设置下载延时，延长每个下载之间的时间间隔
#DOWNLOAD_DELAY = 3
# The download delay setting will honor only one of:
# 设置同一域名和同一 IP 的并发数
#CONCURRENT_REQUESTS_PER_DOMAIN = 16
#CONCURRENT_REQUESTS_PER_IP = 16
```

24.4　定义存储字段

项目所爬取的数据分为房屋信息和小区信息，两者之间存在一对多的数据关系，一个小区可以同时出售多套房屋，而房屋只能属于某个小区。根据这个数据关系，我们在 items.py 定义了两个字段，分别代表房屋信息和小区信息，代码如下：

```
import scrapy
class LianjiaItem(scrapy.Item):
    villageInfo = scrapy.Field()
    houseInfo = scrapy.Field()
```

上述代码分别定义 villageInfo 和 houseInfo 字段，每个字段所存储的数据以字典格式表示，字典的每个键值对代表某套房屋或某个小区的具体信息，我们把房屋和小区的具体信息以表 24-2 表示。

表 24-2　房屋和小区信息表

存储字段	字段命名
房屋信息	
房屋编号	house_hid
建筑面积	acreage
房屋户型	type
所在楼层	high
户型结构	structure
套内面积	innerAcreage
建筑类型	style
房屋朝向	orientation
建筑结构	framework
装修情况	renovation
梯户比例	proportion
配备电梯	elevator
产权年限	years
售价	price
每平方售价	unitPrice
挂牌时间	listingTime
交易权属	tradingRights
上次交易	lastTransaction
房屋用途	use
房屋年限	life
产权所属	belong
地址链接	url
小区编号	region_rid

（续表）

存储字段	字段命名
小区信息	
小区编号	region_rid
小区名称	name
小区位置	area
建成日期	buildYear
建筑类型	buildType
物业费用	buildCost
物业公司	costCompany
开发商	developers
楼栋总数	buildCount
房屋总数	houseCount
附近门店	nearby

从表上的数据可以看到，房屋信息和小区信息包含了多个数据，如果这些数据在 items.py 里逐一定义，就会增加项目文件 items.py、pipelines.py 和 houseSpider.py 的代码量，因此将这些数据以字典格式表示可以精简项目的代码量。

24.5 定义管道类

从表 24-2 的存储字段得知，房屋信息共有 23 个字段，小区信息共有 11 个字段。在 pipelines.py 文件里分别对这些字段进行定义，由 SQLAlchemy 实现定义过程和构建数据库映射关系，定义映射类 houseInfo 和 villageInfo 分别对应房屋信息和小区信息，代码如下：

```
from sqlalchemy import *
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative base
from datetime import datetime
# 导入 setting 配置信息
from scrapy.conf import settings

# 定义映射类
Base = declarative base()
# 小区信息表
class villageInfo(Base):
      tablename   = 'villageInfo'
    id = Column(Integer(), primary key=True)
    region_rid = Column(String(100), comment='小区编号')
    name = Column(String(100), comment='小区名称')
    area = Column(String(100), comment='小区位置')
    buildYear = Column(Text(), comment='建成日期')
    buildType = Column(String(100), comment='建筑类型')
    buildCost = Column(String(100), comment='物业费用')
    costCompany = Column(String(100), comment='物业公司')
```

```
        developers = Column(String(100), comment='开发商')
        buildCount = Column(String(100), comment='楼栋总数')
        houseCount = Column(String(100), comment='房屋总数')
        nearby = Column(String(100), comment='附近门店')
        log date = Column(DateTime(), default=datetime.now,
                        onupdate=datetime.now, comment='记录日期')
    # 房屋信息表
    class houseInfo(Base):
          tablename   = 'houseInfo'
        id = Column(Integer(), primary key=True)
        house_hid = Column(String(100), comment='房屋编号')
        acreage = Column(String(100), comment='建筑面积')
        type = Column(String(100), comment='房屋户型')
        high = Column(String(100), comment='所在楼层')
        structure = Column(String(100), comment='户型结构')
        innerAcreage = Column(String(100), comment='套内面积')
        style = Column(String(100), comment='建筑类型')
        orientation = Column(String(100), comment='房屋朝向')
        framework = Column(String(100), comment='建筑结构')
        renovation = Column(String(100), comment='装修情况')
        proportion = Column(String(100), comment='梯户比例')
        elevator = Column(String(100), comment='配备电梯')
        years = Column(String(100), comment='产权年限')
        price = Column(String(100), comment='售价')
        unitPrice = Column(String(100), comment='每平方售价')
        listingTime = Column(String(100), comment='挂牌时间')
        tradingRights = Column(String(100), comment='交易权属')
        lastTransaction = Column(String(100), comment='上次交易')
        use = Column(String(100), comment='房屋用途')
        life = Column(String(100), comment='房屋年限')
        belong = Column(String(100), comment='产权所属')
        url = Column(String(100), comment='地址链接')
        region rid = Column(String(100), comment='小区编号')
        log date = Column(DateTime(), default=datetime.now,
                        onupdate=datetime.now, comment='记录日期')
```

映射类 houseInfo 和 villageInfo 除了定义存储字段之外，还定义了字段 id 和 log_date，分别代表数据表的主键 id 和数据的记录日期，字段 id 会在数据插入的时候自动生成一个递增的整数；字段 log_date 会在数据插入或更新的时候记录当前操作的时间。

除了定义映射类 houseInfo 和 villageInfo 之外，项目文件 pipelines.py 最主要的是实现数据存储功能，这个功能由 LianjiaPipeline 类实现，实现代码如下：

```
    class LianjiaPipeline(object):
        def   init  (self):
            # 初始化，连接数据库
            conntion = settings['MYSQL CONNECTION']
            engine = create engine(conntion,echo=False,pool size=2000)
            DBSession = sessionmaker(bind=engine)
            self.SQLsession = DBSession()
            # 创建数据表
            Base.metadata.create_all(engine)
```

```
    # 写入房屋信息
    def house_db(self, info):
        house_hid = info['house_hid']
        # 判断是否已存在记录
        temp = self.SQLsession.query(houseInfo).
                filter_by(house_hid=house_hid).first()
        if temp:
            temp.acreage = info.get('acreage', '')
            temp.type = info.get('type', '')
            temp.high = info.get('high', '')
            temp.structure = info.get('structure', '')
            temp.innerAcreage = info.get('innerAcreage', '')
            temp.style = info.get('style', '')
            temp.orientation = info.get('orientation', '')
            temp.framework = info.get('framework', '')
            temp.renovation = info.get('renovation', '')
            temp.proportion = info.get('proportion', '')
            temp.elevator = info.get('elevator', '')
            temp.years = info.get('years', '')
            temp.price = info.get('price', '')
            temp.unitPrice = info.get('unitPrice', '')
            temp.listingTime = info.get('listingTime', '')
            temp.tradingRights = info.get('tradingRights', '')
            temp.lastTransaction = info.get('lastTransaction', '')
            temp.use = info.get('use', '')
            temp.life = info.get('life', '')
            temp.belong = info.get('belong', '')
            temp.url = info.get('url', '')
            temp.region_rid = info.get('region_rid', '')
        else:
            inset_data = houseInfo(
                house_hid=info.get('house_hid', ''),
                acreage=info.get('acreage', ''),
                type=info.get('type', ''),
                high=info.get('high', ''),
                structure=info.get('structure', ''),
                innerAcreage=info.get('innerAcreage', ''),
                style=info.get('style', ''),
                orientation=info.get('orientation', ''),
                framework=info.get('framework', ''),
                renovation=info.get('renovation', ''),
                proportion=info.get('proportion', ''),
                elevator=info.get('elevator', ''),
                years=info.get('years', ''),
                price=info.get('price', ''),
                unitPrice=info.get('unitPrice', ''),
                listingTime=info.get('listingTime', ''),
                tradingRights=info.get('tradingRights', ''),
                lastTransaction=info.get('lastTransaction', ''),
                use=info.get('use', ''),
                life=info.get('life', ''),
                belong=info.get('belong', ''),
                url=info.get('url', ''),
                region_rid=info.get('region_rid', ''),
```

```
            )
            self.SQLsession.add(inset_data)
        self.SQLsession.commit()

    # 写入小区信息
    def village_db(self, info):
        region_rid = info['region_rid']
        # 判断是否已存在记录
        temp = self.SQLsession.query(villageInfo).
                filter_by(region_rid=region_rid).first()
        if temp:
            temp.name = info.get('name')
            temp.area = info.get('area', '')
            temp.buildYear = info.get('buildYear', '')
            temp.buildType = info.get('buildType', '')
            temp.buildCost = info.get('buildCost', '')
            temp.costCompany = info.get('costCompany', '')
            temp.developers = info.get('developers', '')
            temp.buildCount = info.get('buildCount', '')
            temp.houseCount = info.get('houseCount', '')
            temp.nearby = info.get('nearby', '')
        else:
            inset_data = villageInfo(
                region_rid=info.get('region_rid', ''),
                name=info.get('name'),
                area=info.get('area', ''),
                buildYear=info.get('buildYear', ''),
                buildType=info.get('buildType', ''),
                buildCost=info.get('buildCost', ''),
                costCompany=info.get('costCompany', ''),
                developers=info.get('developers', ''),
                buildCount=info.get('buildCount', ''),
                houseCount=info.get('houseCount', ''),
                nearby=info.get('nearby', '')
            )
            self.SQLsession.add(inset_data)
        self.SQLsession.commit()

    # 入库处理
    def process_item(self, item, spider):
        self.house_db(item['houseInfo'])
        self.village_db(item['villageInfo'])
        return item
```

数据存储类 LianjiaPipeline 重写初始化方法__init__()和定义类方法 house_db()、village_db()和 process_item()，各个方法的功能说明如下：

- 初始化方法__init__()是读取配置文件 settings.py 的配置属性 MYSQL_CONNECTION，用于 SQLAlchemy 连接 MySQL 数据库，将数据库连接对象设为类属性 SQLsession，便于类方法调用，从而实现数据库操作。
- 类方法 house_db()的参数 info 代表房屋信息并以字典格式传入。house_db()首先获取房屋 ID，用来标记房屋的唯一性，以房屋 ID 作为查询条件，对数据表 houseInfo 进行查询，如果数据

表已存在该房屋信息，则对数据表的数据进行更新操作，反之则对数据库插入当前数据。

- 类方法 village_db()与 house_db()实现的功能是相同的，其中 village_db()的参数 info 代表小区信息并以字典格式传入；然后以小区 ID 作为查询条件，查询的数据表为 villageInfo。
- 类方法 process_item()的参数 item 是由项目文件 items.py 的 LianjiaItem 类实例化所生成的对象，它包含了房屋信息和小区信息，这些信息是由 spider 程序写入的；process_item()调用了 village_db()和 house_db()方法，并对调用的方法分别传入房屋信息和小区信息，从而实现数据入库处理。

24.6 编写爬虫规则

从项目分析得知，爬取的网页分为房屋列表页、房屋详情页和小区详情页，其中房屋列表页设有分页功能，需要两次 HTTP 请求才能读取所有房屋信息；另外两个页面的数据只需一次 HTTP 请求即可获取，因此本项目的 Spider 共有 4 个类方法，所实现的功能说明如下：

- 遍历访问房屋列表页的总页数。
- 获取房屋列表页的每一页的房屋信息。
- 爬取每套房屋的详细信息。
- 爬取房屋所在小区的详细信息。

根据上述功能说明，在项目文件 houseSpider.py 编写相应的功能代码，如下所示：

```
from lianjia.items import LianjiaItem
from scrapy.selector import Selector
from scrapy.spider import Spider, Request
import configparser, json

class HouseSpider(Spider):
    # 属性 name 必须设置，而且是唯一命名，用于运行爬虫
    name = "House"
    # 设置允许访问域名
    allowed_domains = ["lianjia.com"]
    # 设置 URL
    start_urls='https://%s.lianjia.com/ershoufang/pg%s/'
    # 重写 start_requests
    def start_requests(self):
        # 读取配置文件，获取电影 ID 并生成列表
        conf = configparser.ConfigParser()
        domainList = []
        conf.read(self.settings.get('CONF'))
        temp = conf['LJ']
        if 'domain' in temp.keys():
            domainList = conf['LJ']['domain'].split(',')
        # 遍历各个城市的域名
        for d in domainList:
            self.domain = d
            headers = self.settings.get('DEFAULT_REQUEST_HEADERS')
            headers['Host'] = self.domain + '.lianjia.com'
```

```
            headers['Upgrade-Insecure-Requests'] = '1'
            # 遍历房屋列表页的总页数
            for p in range(100):
                url = self.start_urls %(self.domain, str(p+1))
                yield Request(url=url, headers=headers,
                                 meta={'headers': headers},
                                 callback=self.pageInfo)

    def pageInfo(self, response):
        sel = Selector(response)
        headers = response.meta['headers']
        houseURL = sel.xpath('//ul[@class="sellListContent"]/li')
        for u in houseURL:
            url = ''.join(u.xpath('.//div[@class="title"]//
                    a//@href').extract()).strip()
            yield Request(url=url,headers=headers,callback=self.housePage)

    def housePage(self, response):
        houseInfo = {}
        villageInfo = {}
        sel = Selector(response)
        # 房屋信息
        houseInfo['url'] = response.url
        houseInfo['house hid']=response.url.split('/')[-1].split('.')[0]
        houseInfo['price'] = ''.join(sel.xpath('//span[@class=
                                  "total"]//text()').extract())+ ''.join(
                                  sel.xpath('//span[@class="unit"]//span//
                                  text()').extract())
        houseInfo['unitPrice'] = ''.join(sel.xpath('//span[@class=
                                      "unitPriceValue"]//text()').extract())
        baseInfo = sel.xpath('//div[@class="base"]//li')
        # 基本信息
        for b in baseInfo:
            i = ''.join(b.xpath('.//text()').extract())
            if '房屋户型' in str(i):
                houseInfo['type']=i.replace('房屋户型', '')
            elif '所在楼层' in str(i):
                houseInfo['high']=i.replace('所在楼层', '')
            elif '建筑面积' in str(i):
                houseInfo['acreage']=i.replace('建筑面积', '')
            elif '户型结构' in str(i):
                houseInfo['structure']=i.replace('户型结构', '')
            elif '套内面积' in str(i):
                houseInfo['innerAcreage']=i.replace('套内面积','')
            elif '建筑类型' in str(i):
                houseInfo['style']=i.replace('建筑类型', '')
            elif '房屋朝向' in str(i):
                houseInfo['orientation']=i.replace('房屋朝向','')
            elif '建筑结构' in str(i):
                houseInfo['framework']=i.replace('建筑结构', '')
            elif '装修情况' in str(i):
                houseInfo['renovation']=i.replace('装修情况', '')
            elif '梯户比例' in str(i):
```

```
                houseInfo['proportion']=i.replace('梯户比例', '')
            elif '配备电梯' in str(i):
                houseInfo['elevator']=i.replace('配备电梯', '')
            elif '产权年限' in str(i):
                houseInfo['years']=i.replace('产权年限', '')
        # 交易信息
        transaction = sel.xpath('//div[@class="transaction"]//li')
        for t in transaction:
            i = ''.join(t.xpath('.//span//text()').extract())
            if '挂牌时间' in str(i):
                houseInfo['listingTime']=i.replace('挂牌时间','').strip()
            elif '交易权属' in str(i):
                houseInfo['tradingRights']=i.replace('交易权属','').strip()
            elif '上次交易' in str(i):
                houseInfo['lastTransaction']=i.replace('上次交易','').strip()
            elif '房屋用途' in str(i):
                houseInfo['use']=i.replace('房屋用途','').strip()
            elif '房屋年限' in str(i):
                houseInfo['life']=i.replace('房屋年限','').strip()
            elif '产权所属' in str(i):
                houseInfo['belong']=i.replace('产权所属','').strip()
        # 小区信息
        villageInfo['region rid'] = ''.join(sel.xpath('//a[@class=
                                     "info "]//@href').extract()).
                                     split('xiaoqu/')[1].replace('/','')
        houseInfo['region rid'] = villageInfo['region rid']
        villageInfo['area'] = ''.join(sel.xpath('//div[@class=
                              "areaName"]//a//text()').extract())
        villageURL = 'https://%s.lianjia.com/xiaoqu/%s/'
                      %(self.domain, villageInfo['region rid'])
        # 构建请求头
        headers = self.settings.get('DEFAULT REQUEST HEADERS')
        headers['Host'] = self.domain + '.lianjia.com'
        headers['X-Requested-With'] = 'XMLHttpRequest'
        headers['Referer'] = houseInfo['url']
        yield Request(url=villageURL, headers=headers,
                      meta={'houseInfo': houseInfo,
                            'villageInfo': villageInfo},
                      callback=self.villagePage,dont filter=True)

    def villagePage(self, response):
        houseInfo = response.meta['houseInfo']
        villageInfo = response.meta['villageInfo']
        sel = Selector(response)
        villageInfo['name'] = ''.join(sel.xpath('//h1[@class=
                              "detailTitle"]//text()').extract())
        villageInfo['area'] = ''.join(sel.xpath('//div[@class=
                              "detailDesc"]//text()').extract())
        info = sel.xpath('//div[@class="xiaoquInfo"]//
                span[@class="xiaoquInfoContent"]')
        villageInfo['buildYear'] = ''.join(info[0].xpath('.//text()').
                                   extract())
        villageInfo['buildType'] = ''.join(info[1].xpath('.//text()').
                                     extract())
```

```
        villageInfo['buildCost'] = ''.join(info[2].xpath('.//text()').
                                   extract())
        villageInfo['costCompany'] = ''.join(info[3].xpath('.//text()').
                                     extract())
        villageInfo['developers'] = ''.join(info[4].xpath('.//text()').
                                   extract())
        villageInfo['buildCount'] = ''.join(info[5].xpath('.//text()').
                                   extract())
        villageInfo['houseCount'] = ''.join(info[6].xpath('.//text()').
                                   extract())
        villageInfo['nearby']=''.join(info[7].xpath('.//text()').extract())
        # 入库处理
        item = LianjiaItem()
        item['houseInfo'] = houseInfo
        item['villageInfo'] = villageInfo
        yield item
```

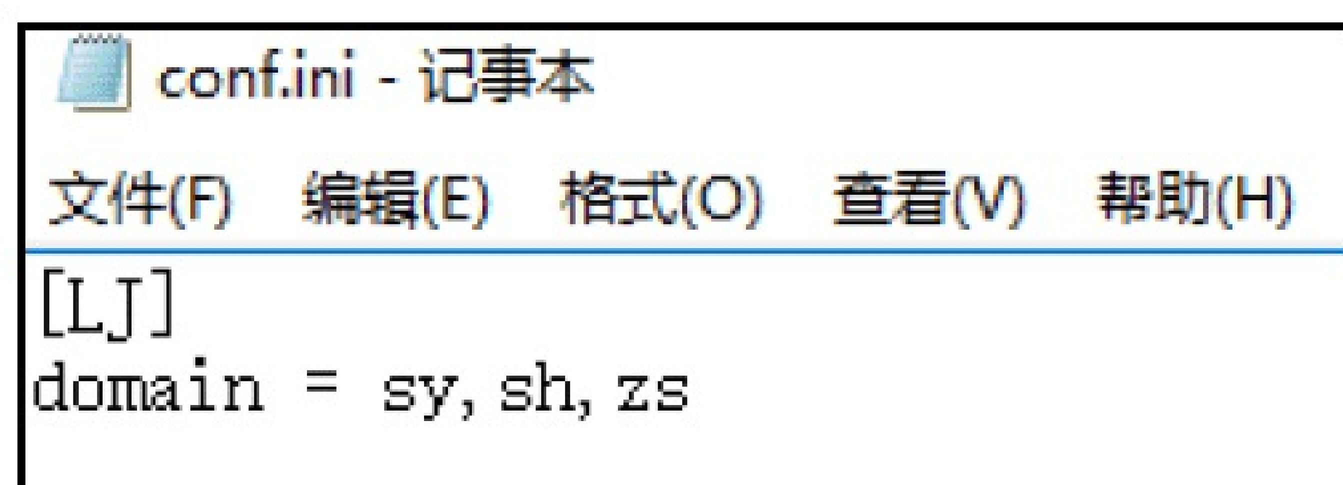
上述代码定义了 4 个类方法，分别是 start_requests()、pageInfo()、housePage()和 villagePage()，每个方法所实现的功能说明如下：

- start_requests()是重写父类 Spider 的方法，首先读取配置文件 conf.ini 的城市域名信息，域名信息是构建请求头的 Host 属性和三个页面的 URL 地址。请求头的 Host 属性是必须设置的属性之一，它代表网站的站点信息，如果 URL 地址的域名是上海，而 Host 属性值为深圳，在访问该 URL 的时候会出现无法访问的情况，这是较为常见的反爬虫机制。
- pageInfo()是处理 start_requests()发送 HTTP 请求的响应内容，从响应内容获取房屋列表页的房屋详情页的 URL 地址，最后对这些 URL 地址发送 HTTP 请求，请求头是由 start_requests()的参数 meta 传递所得。
- housePage()首先处理 pageInfo()发送 HTTP 请求的响应内容，从响应内容提取房屋详细信息以及小区详情页的 URL 地址；然后重新构建请求头，设置 X-Requested-With 和 Referer 属性，用于访问小区详情页的 URL 地址；最后设置参数 meta，将房屋信息 houseInfo 和小区信息 villageInfo 传递给回调方法 villagePage()。
- villagePage()是处理 housePage()发送 HTTP 请求的响应内容，爬取小区详细信息并写入字典 villageInfo；然后将 items.py 的 LianjiaItem 类实例化，生成 item 对象；最后将房屋信息 houseInfo 和小区信息 villageInfo 写入 item 对象，传递给 pipelines.py 的 LianjiaPipeline 类，实现数据入库处理。

这 4 个类方法所实现的功能得知，每个方法之间存在一定的关联及数据交互，比如 start_requests()发送的 HTTP 请求是由回调方法 pageInfo()处理响应内容，参数 meta 的参数值也是传给回调方法使用。

从 spider 程序读取配置文件 conf.ini 的方式可以知道，每个城市的域名是以英文逗号进行拼接，然后赋值给配置属性 domainList，如图 24-6 所示。

在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl House，启动并运行项目 lianjia。项目运行完成后，打开 spiderdb 数据库分别查看数据表 houseinfo 和 villageinfo 的数据信息，如图 24-7 所示。



conf.ini - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
[LJ]
domain = sy, sh, zs
```

图 24-6　配置文件

villageinfo @spiderdb (MyDb) - 表

文件 编辑 查看 窗口 帮助

开始事务 文本 筛选 排序 导入 导出

id	region_rid	name	area	buildYear	buildType
1	311335066599	金地名京三期	(铁西铁西区)小北一中路27号	2013年建成	板楼
2	311105338844	SR国际新城	(浑南浑南新区)浑河堡街	2009年建成	塔楼/塔板结合

houseinfo @spiderdb (MyDb) - 表

文件 编辑 查看 窗口 帮助

开始事务 文本 筛选 排序 导入 导出

id	house_hid	acreage	type	high	structure	innerAcreage
1	102100303888	89.78㎡	2室2厅1厨1卫	低楼层 (共24层)	平层	暂无数据
2	102100928000	158.53㎡	3室1厅1厨2卫	中楼层 (共30层)	平层	暂无数据

图 24-7　数据表信息

24.7　本 章 小 结

本章介绍了使用 Scrapy 编写爬取链家二手房信息的程序，通过本章的学习，读者应当掌握以下技能：

1. 设计爬虫爬取方向

- 根据房屋列表页的 URL 地址构造规律，动态设置 URL 末端的页数来获取全部房屋详情页的 URL 地址。这个获取过程涉及两个循环：页数循环和每页的房屋列表循环；前者是循环 100 页的房屋列表，后者则是获取每页房屋列表的房屋详情页 URL 地址。
- 房屋详情页的 URL 地址末端的一串数字代表房屋 ID，用于标记房屋的唯一性。在房屋详情页里除了爬取房屋的基本信息之外，还能爬取小区详情页的 URL 地址，从而访问小区详情页爬取小区信息。
- 小区详情页的 URL 地址末端的一串数字代表小区 ID，用于标记小区的唯一性。在小区详情页爬取小区基本信息之外，还要将小区和房屋的数据相互关联，因为会出现一个小区有多套房屋出售的情况。

2. 设计数据存储功能

数据存储类 LianjiaPipeline 重写初始化方法__init__()和定义类方法 house_db()、village_db()和 process_item()，各个方法的功能说明如下：

- 初始化方法__init__()是读取配置文件 settings.py 的配置属性 MYSQL_CONNECTION，用于 SQLAlchemy 连接 MySQL 数据库，将数据库连接对象设为类属性 SQLsession，便于类方法调用，从而实现数据库操作。
- 类方法 house_db()的参数 info 代表房屋信息并以字典格式传入。house_db()首先获取房屋 ID，用于标记房屋的唯一性，以房屋 ID 作为查询条件，对数据表 houseInfo 进行查询，如果数据表已存在该房屋信息，则对数据表的数据进行更新操作，反之则对数据库插入当前数据。
- 类方法 village_db()与 house_db()实现的功能相同，其中 village_db()的参数 info 代表小区信息并以字典格式传入；然后以小区 ID 作为查询条件，查询的数据表为 villageInfo。
- 类方法 process_item()的参数 item 是由项目文件 items.py 的 LianjiaItem 类实例化所生成的对象，它包含了房屋信息和小区信息，这些信息是由 spider 程序写入的；process_item()调用了 village_db()和 house_db()方法，并对调用的方法分别传入房屋信息和小区信息，从而实现数据入库处理。

3. 爬虫规则 Spider

爬取的网页分为房屋列表页、房屋详情页和小区详情页，其中房屋列表页设有分页功能，需要两次 HTTP 请求才能读取所有房屋的信息；另外两个页面的数据只需一次 HTTP 请求即可获取，因此本项目的 Spider 共有 4 个类方法，所实现的功能说明如下：

- 遍历访问房屋列表页的总页数。
- 获取房屋列表页的每一页的房屋信息。
- 爬取每套房屋的详细信息。
- 爬取房屋所在小区的详细信息。

第 25 章

实战：QQ 音乐全站爬取

25.1 项目分析

在第 18 章，我们介绍了使用 Requests 爬取 QQ 音乐，本章将使用 Scrapy 爬取 QQ 音乐，实现与第 18 章相同的功能，并且沿用第 18 章的爬虫规则实现项目开发。

在歌手列表（https://y.qq.com/portal/singer_list.html）中按照字母类别对歌手分类，遍历每个分类下的每位歌手页面，然后获取每位歌手页面的全部歌曲信息。根据该设计方案列出遍历次数：

（1）遍历每位歌手的歌曲页数。
（2）遍历每个字母分类的每页歌手信息。
（3）遍历每个字母分类的歌手总页数。
（4）遍历 26 个字母和特殊符号分类的歌手列表。

在功能上至少需要实现 4 次遍历，但在实际开发中往往比这个次数要多。统计遍历次数，主要是能让开发者对项目开发有一个整体设计逻辑。项目开发使用模块化设计思想，整个项目模块的划分如下：

（1）歌曲下载。
（2）歌手信息和歌曲信息。
（3）字母分类下的歌手列表。
（4）全站歌手列表。

按照上述方案，Scrapy 爬取 QQ 音乐的开发顺序如下。

（1）setting.py：配置爬虫信息，如请求头、数据库信息、文件保存路径。
（2）items.py：定义存储数据对象，主要存储歌曲相关信息。
（3）pipelines.py：数据存储，实现歌曲信息入库和歌曲下载。
（4）spiders 文件夹（musicSpider.py）：编写爬虫规则。

由于项目的爬取对象和爬取策略与第 18 章相同，所以本章不再对 QQ 音乐网站进行分析，对爬取网站架构不清晰的读者可阅读第 18 章的有关内容。

25.2　项目创建与配置

25.2.1　项目创建

首先创建 Scrapy 爬虫项目，命名为 music，打开 CMD 命令提示符窗口，将当前的路径切换到其他磁盘，然后输入创建指令：

```
scrapy startproject music
```

项目创建完成后，在项目中的 spiders 文件夹里创建 musicSpider.py 文件，该文件用于实现 Spider 功能。最后在 PyCharm 中打开项目所在的文件夹，目录结构如图 25-1 所示。

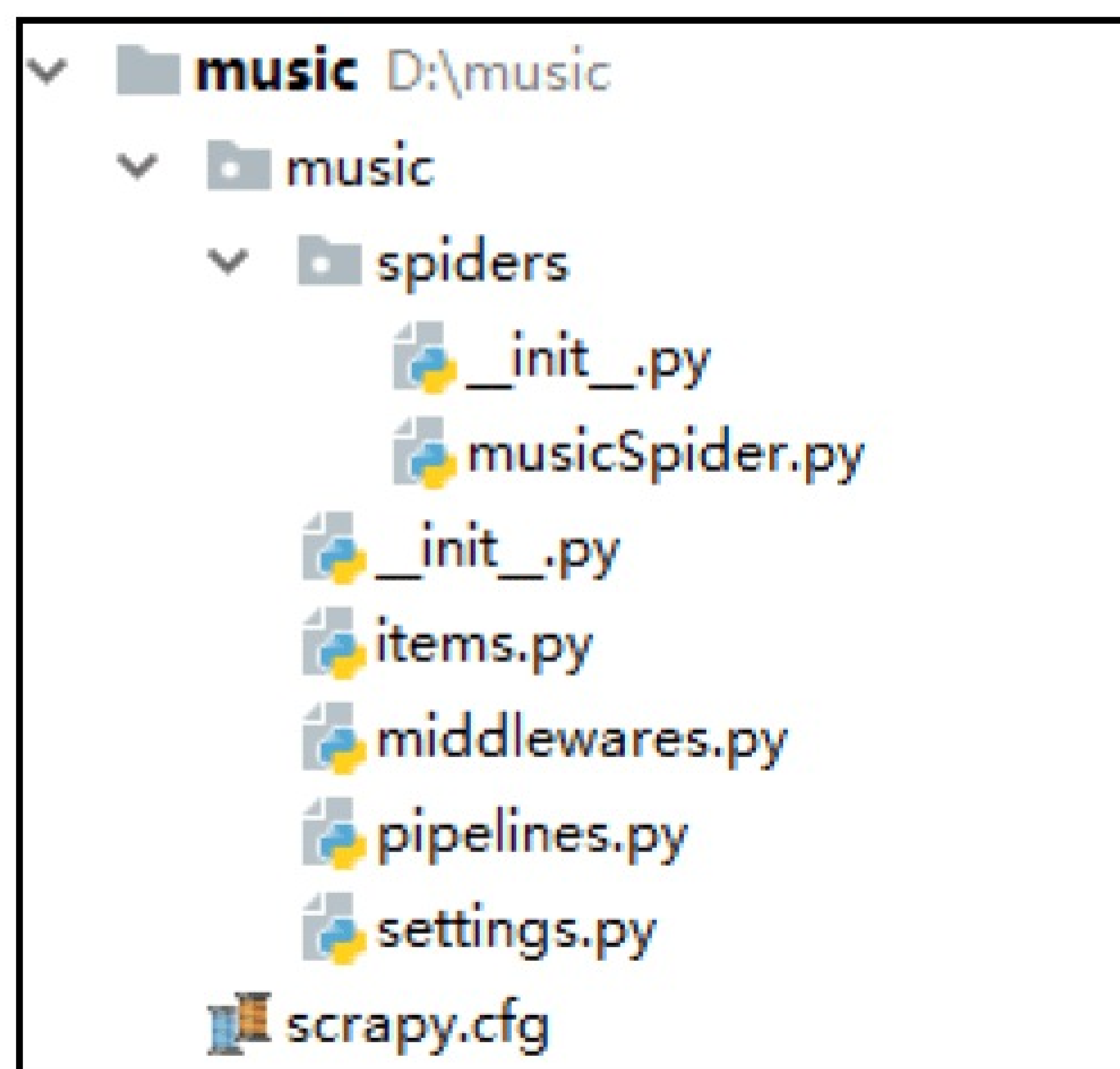


图 25-1　目录结构

25.2.2　项目配置

完成项目创建后，接下来就进入项目开发。按照制定的开发顺序，首先完成项目配置的开发，打开项目的配置文件 setting.py，由于文件在创建的时候已有较多的注释代码，因此此处只列出项目所需的代码内容。其代码如下：

```
BOT_NAME = 'music'
SPIDER_MODULES = ['music.spiders']
NEWSPIDER_MODULE = 'music.spiders'
# 设置 False
ROBOTSTXT_OBEY = False
# 定义请求头
DEFAULT_REQUEST_HEADERS = {
  'Accept': 'text/html,application/xhtml+xml,
```

```
                application/xml;q=0.9,*/*;q=0.8',
      'Accept-Language': 'en',
      'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                     AppleWebKit/537.36 (KHTML, like Gecko)
                     Chrome/69.0.3497.100 Safari/537.36'
    }
    # 设置管道功能类
    ITEM_PIPELINES = {
       'music.pipelines.MusicPipeline': 300,
       'music.pipelines.DownloadMusicPipeline': 300,
    }
    # 数据库连接信息
    MYSQL_CONNECTION = 'mysql+pymysql://root:1234@
                        localhost:3306/music_db?charset=utf8'
    # 设置歌曲的保存路径
    FILES_STORE = r'D:\\full\\'
```

从上述代码看出，项目分别对 Item Pipelines、数据库信息、请求头和文件保存路径进行配置，各个配置说明如下。

- Item Pipelines：创建项目时，默认配置了类 MusicPipeline。在此项目中，需要添加一个下载类 DownloadMusicPipeline，该类继承自父类 FilesPipeline，主要实现歌曲下载功能。
- 数据库信息：该配置属于自定义配置信息，变量 MYSQL_CONNECTION 以字符串格式表示，变量值是 SQLAlchemy 连接数据库语句。数据库系统为本地数据库系统，数据库为 music_db。
- 请求头：配置默认的请求头内容，如果项目中发送 HTTP 请求并没有指定请求头，就默认使用该配置作为请求头。
- 文件保存路径：属于自定义配置信息，变量 FILES_STORE 为字符串格式，内容是系统有效路径，主要用于歌曲下载的保存路径。

除此之外，还可以配置并发数和下载延时等相关信息。本节使用默认配置即可，读者可根据以下代码自行配置：

```
    # Configure maximum concurrent requests performed by Scrapy (default: 16)
    # 设置并发数，Scrapy 默认同一时间可并发 16 个请求
    #CONCURRENT_REQUESTS = 32
    # Configure a delay for requests for the same website (default: 0)
    # See
http://scrapy.readthedocs.org/en/latest/topics/settings.html#download-delay
    # See also autothrottle settings and docs
    # 设置下载延时，延长每个下载之间的时间间隔
    #DOWNLOAD_DELAY = 3
    # The download delay setting will honor only one of:
    # 设置同一域名和同一 IP 的并发数
    #CONCURRENT_REQUESTS_PER_DOMAIN = 16
    #CONCURRENT_REQUESTS_PER_IP = 16
```

25.3　定义存储字段和管道类

25.3.1　定义存储字段

项目的 items.py 主要用于定义歌曲信息的存储对象，衔接爬虫规则 Spider 和管道 Item Pipelines，使两者之间的数据交互传递。根据第 18 章数据存储部分的介绍，我们将需要存储的歌曲信息以表 25-1 所示。

表 25-1　song 数据表

字段	说明
song_id	主键
song_name	歌名
song_ablum	所属专辑
song_interval	歌曲播放时长
song_songmid	歌曲 mid
song_singer	演唱歌手
song_url	歌曲播放链接

字段 song_id 是数据表的主键并且数据是自动生成，在爬取的数据中并不存在，所以在 items.py 无须定义该字段。而歌曲下载链接 song_url，除了写入 MySQL 数据库之外，还用于管道类 DownloadMusicPipeline 实现歌曲下载。综合分析，项目文件 items.py 的代码如下：

```
import scrapy
class MusicItem(scrapy.Item):
    song_name = scrapy.Field()
    song_ablum = scrapy.Field()
    song_interval = scrapy.Field()
    song_songmid = scrapy.Field()
    song_singer = scrapy.Field()
    song_url = scrapy.Field()
    pass
```

25.3.2　定义管道类

根据 items.py 的存储字段来实现数据存储功能，数据存储功能是在项目文件 pipelines.py 里实现，该文件主要实现两个功能：歌曲信息入库和歌曲下载。

1. 歌曲信息入库

歌曲信息入库主要由 MusicPipeline 实现，该类在创建项目时已自动生成，但具体的存储过程还需要开发者自行编写相关代码，数据存储的代码如下：

```
import scrapy
from scrapy.pipelines.files import FilesPipeline
```

```
from sqlalchemy import *
from sqlalchemy.orm import sessionmaker
from sqlalchemy.ext.declarative import declarative base
from scrapy.conf import settings

# SQLAlchemy 映射数据表
Base = declarative base()
class song(Base):
    # 表名
      tablename   = 'song'
    # 字段，属性
    song id = Column(Integer, primary key=True)
    song name = Column(String(200))
    song ablum = Column(String(200))
    song interval = Column(String(200))
    song songmid = Column(String(200))
    song_singer = Column(String(200))
    song url = Column(String(2000))

# 数据入库
class MusicPipeline(object):
    def   init   (self):
        # 获取配置信息 setting.py 的数据库连接
        connection = settings['MYSQL CONNECTION']
        # 连接数据库
        engine = create engine(connection, echo=False)
        # 创建会话对象，用于数据表的操作
        DBSession = sessionmaker(bind=engine)
        self.SQLsession = DBSession()
        # 创建数据表
        Base.metadata.create all(engine)

    def process item(self, item, spider):
        song songmid = item['song songmid']
        temp = self.SQLsession.query(song).filter by(
                song songmid=song songmid).first()
        # 已存在的数据做更新处理
        if temp:
            temp.song name = item['song name'],
            temp.song ablum = item['song ablum'],
            temp.song interval = item['song interval'],
            temp.song singer = item['song singer'],
            temp.song_url = item['song_url']
        # 判断歌曲是否有播放版权
        elif 'vkey' in item['song url']:
            data = song(
                song name=item['song name'],
                song ablum=item['song ablum'],
                song interval=item['song interval'],
                song songmid=item['song songmid'],
                song_singer=item['song_singer'],
                song url=item['song url']
            )
            self.SQLsession.add(data)
```

```
        self.SQLsession.commit()
        return item
```

上述代码主要实现三个功能：定义数据表映射类、重写 MusicPipeline 类的初始化方法__init__()和定义类方法 process_item()，具体说明如下。

- 数据表映射类用于映射数据表 song，类属性是数据表的字段。映射类只是将数据表以对象的形式表示，在实际上，数据表和映射类还没真正实现连接。
- 初始化方法__init__()用于读取配置文件 settings.py 的配置属性 MYSQL_CONNECTION，用于 SQLAlchemy 连接 MySQL 数据库，将数据库连接对象设为类属性 SQLsession，便于类方法调用，从而实现数据库操作。
- 类方法 process_item()是参数 items 的数据进行入库处理，从数据写入方式来看，参数 items 代表一首歌曲的信息，也就是每爬取一首歌曲，就会执行一次歌曲入库和下载。但是并不是每一首歌都会写入数据库，因为有些歌曲的版权问题，导致无法播放。通过判断歌曲播放的 URL 地址来决定歌曲是否入库处理，因为歌曲播放的 URL 地址没有请求参数就能说明歌曲没有播放版权，如图 25-2 所示。

	歌曲	专辑
1	All These Lovers	Life.Love.Work.Dreams
2	How Else	Life.Love.Work.Dreams
3	Everything In You	Life.Love.Work.Dreams
4	Goodnight In The Morning	Life.Love.Work.Dreams

图 25-2　歌曲版权问题

2. 歌曲下载

完成歌曲信息入库后，还要实现歌曲下载功能，歌曲下载是由 DownloadMusicPipeline 实现，该类属于自定义类，它是继承父类 FilesPipeline，代码如下：

```
# 下载文件
class DownloadMusicPipeline(FilesPipeline):
    # 重写 get media requests
    def get media requests(self, item, info):
        # 设置文件名
        file name = item['song songmid'] + '.m4a'
        yield scrapy.Request(item['song url'], meta={'name': file name})

    # 重写 file_path，命名文件名
    def file path(self, request, response=None, info=None):
        file name = settings['FILES STORE'] + (request.meta['name'])
        return file_name
```

歌曲下载由类方法 get_media_requests()和 file_path()共同实现，两者都是从父类 FilesPipeline

继承并重写的。父类 FilesPipeline 有一套完善的下载机制，但很多时候并不符合各种各样的下载需求，所以大多数情况下都是通过类的继承和重写的方式实现需求化下载。

- get_media_requests()：参数 items 代表一首歌曲的信息，从 items 对象获取歌曲的 songmid 作为文件名，然后将文件名作为 scrapy.Request 的 meta 参数传递给 file_path()。
- file_path()：接收 get_media_requests()传递的参数 meta，并读取 setting.py 里面的文件路径配置信息，组合成一个完整的文件路径，最后 DownloadMusicPipeline 将下载的文件以 file_path()返回值作为文件名。

25.4 编写爬虫规则

爬虫规则是整个项目中的难点，同时也是代码量最多的一个功能。根据第 18 章实现的功能发现，整个程序共发送了 6 个不同的请求。对于 Scrapy 的 Spider 来说，一个 HTTP 请求是以一个类方法表示，因此本项目的 Spider 共有 6 个类方法，相应的功能有：

（1）歌手字母分类 A～Z 和特殊符号#。

（2）获取每个字母分类下的每页歌手。

（3）获取每一个歌手信息。

（4）获取歌手的每一页歌曲。

（5）获取每一页的每一首歌曲信息。

（6）每一首歌曲信息。

综合上述分析，项目文件 musicSpider.py 的爬虫代码如下所示：

```
import scrapy
import json
import math, requests
from music.items import MusicItem
from scrapy.spider import Spider
from urllib.parse import quote

class QQMusic(Spider):
    name = 'Music'
    allowed domains = ['qq.com']
    # start urls 是歌手列表 URL
    start_urls = [
        'https://u.y.qq.com/cgi-bin/musicu.fcg?loginUin=0&
         hostUin=0&format=jsonp&inCharset=utf8&outCharset=utf-8
         &notice=0&platform=yqq&needNewCode=0&data=%7B%22comm%2
         2%3A%7B%22ct%22%3A24%2C%22cv%22%3A10000%7D%2C%22singer
         List%22%3A%7B%22module%22%3A%22Music.SingerListServer%
         22%2C%22method%22%3A%22get singer list%22%2C%22param%22
         %3A%7B%22area%22%3A-100%2C%22sex%22%3A-100%2C%22genre%2
         2%3A-100%2C%22index%22%3A',
        '%2C%22sin%22%3A0%2C%22cur page%22%3A1%7D%7D%7D'
        ]
    # 遍历歌手字母分类 A-Z 和特殊符号#
```

```
    def start_requests(self):
        lua = """function main(splash)
          splash:go("https://y.qq.com/")
          splash:wait(3)
          splash:go("https://c.y.qq.com/v8/fcg-bin/fcg_v8_singer
                    _track_cp.fcg?g_tk=5381&jsonpCallback=MusicJs
                    onCallbacksinger_track&loginUin=0&hostUin=0&
                    format=jsonp&inCharset=utf8&outCharset=utf-8&
                    notice=0&platform=yqq&needNewCode=0&singermid
                    =001fNHEf1SFEFN&order=listen&begin=0&num=30&s
                    ongstatus=1")
          splash:wait(3)
          return {
            splash:get_cookies()}
          end"""
        url='http://192.168.99.100:8050/execute?lua_source='+quote(lua)
        response = requests.get(url)
        print(response.json())
        cookie_dict = {}
        for i in response.json()['1']:
            cookie_dict[i['name']] = i['value']
        self.guid = cookie_dict['pgv_pvid']
        self.cookies = cookie_dict
        for index in range(1, 28):
            url = self.start_urls[0] +(str(index))+self.start_urls[1]
            yield scrapy.Request(url, dont_filter=True,
                                 callback=self.get_genre_singer,
                                 meta={'index': index})

    # 获取每个字母分类下的每页歌手
    def get_genre_singer(self, response):
        index = response.meta['index']
        # 从函数 start_requests 得出响应内容，获取总页数
        # str(response.body.decode('utf-8'))
        pagenum = json.loads(response.body.decode('utf-8'))
                            ['singerList']['data']['total']
        # 生成列表
        page_list = [x for x in range(1, pagenum+1)]
        for page in page_list:
            url = 'https://u.y.qq.com/cgi-bin/musicu.fcg?loginUin=
                  0&hostUin=0&format=jsonp&inCharset=utf8&outCharset
                  =utf-8&notice=0&platform=yqq&needNewCode=0&data=
                  %7B%22comm%22%3A%7B%22ct%22%3A24%2C%22cv%22%3A100
                  00%7D%2C%22singerList%22%3A%7B%22module%22%3A%22Mu
                  sic.SingerListServer%22%2C%22method%22%3A%22get_si
                  nger_list%22%2C%22param%22%3A%7B%22area%22%3A-100%
                  2C%22sex%22%3A-100%2C%22genre%22%3A-100%2C%22index%
                  22%3A' + str(index) + '%2C%22sin%22%3A' + str((page
                  -1)*80)+'%2C%22cur_page%22%3A'+str(page)+'%7D%7D%7D'
            # dont_filter 取消重复请求。
            yield scrapy.Request(url, dont_filter=True,
                                 callback=self.get_singer_songs)

    # 获取每一个歌手信息
    def get_singer_songs(self, response):
```

```
        # 获取每个字母分类下的每页歌手的全部信息
        singermid_list = json.loads(response.body.decode('utf-8'))
                                ['singerList']['data']['singerlist']
        for k in singermid_list:
            url = 'https://c.y.qq.com/v8/fcg-bin/fcg_v8_singer_
                   track_cp.fcg?loginUin=0&hostUin=0&singermid=
                   %s&order=listen&begin=0&num=30&songstatus=1'%
                   (k['singer_mid'])
            yield scrapy.Request(url, dont_filter=True,
                                callback=self.get_singer_info,
                                meta={'singermid':k['singer_mid']})

    # 获取歌手的每一页歌曲
    def get_singer_info(self, response):
        # 参数传递获取 singermid
        singermid = response.meta['singermid']
        # 获取歌手的名字，总页数
        singer_info = json.loads(response.body.decode('utf-8'))
        song_singer = singer_info['data']['singer_name']
        songcount = singer_info['data']['total']
        pagecount = math.ceil(int(songcount) / 30)
        for p in range(pagecount):
            url = 'https://c.y.qq.com/v8/fcg-bin/fcg_v8_singer_
                   track_cp.fcg?loginUin=0&hostUin=0&singermid=%s
                   &order=listen&begin=%s&num=30&songstatus=1'%
                   (singermid, p * 30)
            yield scrapy.Request(url, dont_filter=True,
                                callback=self.get_song_info,
                                meta={'song_singer': song_singer})

    # 获取每一页的每一首歌曲信息
    def get_song_info(self, response):
        # 参数传递获取歌手名字
        song_singer = response.meta['song_singer']
        music_data=json.loads(response.body.decode('utf-8'))
                            ['data']['list']
        for i in music_data:
            songmid = i['musicData']['songmid']
            url = 'https://u.y.qq.com/cgi-bin/musicu.fcg?loginUin=0&
                   hostUin=0&format=jsonp&inCharset=utf8&outCharset=
                   utf-8&notice=0&platform=yqq&needNewCode=0&data=
                   %7B%22req%22%3A%7B%22module%22%3A%22CDN.SrfCdnDisp
                   atchServer%22%2C%22method%22%3A%22GetCdnDispatch%2
                   2%2C%22param%22%3A%7B%22guid%22%3A%22'+self.guid+
                   '%22%2C%22calltype%22%3A0%2C%22userip%22%3A%22%22%
                   7D%7D%2C%22req_0%22%3A%7B%22module%22%3A%22vkey.Ge
                   tVkeyServer%22%2C%22method%22%3A%22CgiGetVkey%22%2
                   C%22param%22%3A%7B%22guid%22%3A%22'+self.guid+'%22
                   %2C%22songmid%22%3A%5B%22'+songmid+'%22%5D%2C%22so
                   ngtype%22%3A%5B0%5D%2C%22uin%22%3A%220%22%2C%22log
                   inflag%22%3A1%2C%22platform%22%3A%2220%22%7D%7D%2C
                   %22comm%22%3A%7B%22uin%22%3A0%2C%22format%22%3A%22
                   json%22%2C%22ct%22%3A20%2C%22cv%22%3A0%7D%7D'
            yield scrapy.Request(url, dont_filter=True,
```

```
                                         callback=self.get_data,
                                         meta={'i':i,'song_singer':song_singer},
                                         cookies=self.cookies)

    # 每一首歌曲信息
    def get_data(self, response):
        # 参数传递
        # song_singer 为歌手名字
        # i 为歌曲信息
        song_singer = response.meta['song_singer']
        i = response.meta['i']
        # items.py 文件的类的实例化，用于传递数据给 pipelines.py 实现存储
        items = MusicItem()
        # 获取下载歌曲的 purl
        purl = json.loads(response.body.decode('utf-8'))
                ['req_0']['data']['midurlinfo'][0]['purl']
        # 数据写入 items，用于传递数据给 pipelines.py 实现存储
        items['song_url']='http://isure.stream.qqmusic.qq.com/%s' %(purl)
        items['song_singer'] = song_singer
        items['song_name'] = i['musicData']['songname']
        items['song_ablum'] = i['musicData']['albumname']
        items['song_interval'] = i['musicData']['interval']
        items['song_songmid'] = i['musicData']['songmid']
        yield items
```

上述代码一共定义了 6 个类方法，每个方法所实现的功能如下：

- start_requests()首先获得 start_urls 里面的 URL 信息，然后循环 27 次，分别得到字母 A～Z 特殊符号#传入 URL，生成不同字母分类的 URL 地址。最后对 27 个 URL 发送 GET 请求，由于项目需要对同一个 URL 发送多次请求获取不同的数据，因此 dont_filter=True 是关闭重复访问 URL 的设置；参数 meta 是将 start_requests()的数据传递到回调方法 get_genre_singer()。此外还使用了 Splash 实现网站的 Cookies 获取，由于歌曲下载需要使用 Cookies 里的信息，因此通过 Splash 来获取 Cookies 内容，从而得到歌曲下载的请求参数。
- get_genre_singer()是处理 start_requests()对 27 个 URL 发送 GET 请求的响应内容。首先分别获取 start_requests()传递的 meta 参数和当前分类的总页数（来自于 start_requests()发送 GET 请求的响应内容），使用得到的数据构建新的 URL，其 URL 代表当前分类的每一页的歌手信息，最后对新构建的 URL 发送 GET 请求，回调方法为 get_singer_songs()。
- get_singer_songs()是处理 get_genre_singer()发送 GET 请求的响应内容。其响应内容是获取当前分类的每一页的歌手信息，得到的歌手信息以列表形式表示，通过遍历该列表分别得到每位歌手的 singermid，然后构建新的 URL，其 URL 代表当前歌手的主页面。最后对新的 URL 发送 GET 请求，获取当前歌手的全部信息，回调方法是 get_singer_info()，并传递参数 meta，代表当前歌手的 singermid。
- get_singer_info()是处理 get_singer_songs()发送 GET 请求的响应内容，从响应内容中获取歌手的信息并计算歌曲的总页数，使用得到的信息构建新的 URL，代表当前歌手的每一页歌曲，最后对新构建的 URL 发送 GET 请求，回调方法是 get_song_info()，参数 meta 为歌手姓名。

- get_song_info()是从上一请求的响应内容中获取歌曲信息，歌曲信息以列表形式表示，通过遍历该列表分别获取每一首歌的信息，并使用得到的歌曲信息构建新的 URL，其 URL 用于获取下载歌曲的 vkey 等下载信息，最后对该 URL 发送 GET 请求，回调方法为 get_data()，参数 meta 是歌手和歌曲信息。
- get_data()是最后一个回调方法，主要用于获取歌曲的下载链接和歌曲信息。通过处理上一请求的响应内容得到歌曲下载的信息并构建歌曲下载链接，同时将得到的歌曲信息和新构建的下载链接写入 Items 对象并返回给 Item Pipelines。

从整个 Spider 分析，实现步骤是：全部字母分类歌手列表→当前字母分类歌手列表→当前分类每页的歌手列表→当前分类当前页数的每位歌手→当前歌手的每页歌曲列表→当前歌手的当前歌曲页数的每一首歌曲→获取歌曲信息并下载歌曲。

从实现步骤与第 18 章实现步骤的对比可以发现两者的顺序是相反的，前者是从大到小、从面到点的实现方式；后者是从小到大、从点到面的实现方式。

运行项目之前，还需要开启 Splash 服务器，在电脑上找到 Docker Quickstart Terminal 图标并双击运行。成功启动 Docker 之后，输入 Splash 启动指令 docker run -p 8050:8050 scrapinghub/splash 并按回车键即可，如图 25-3 所示。

```
000@DESKTOP-GOD9Q18 MINGW64 /c/Program Files/Docker Toolbox
$ docker run -p 8050:8050 scrapinghub/splash
2018-11-12 08:18:56+0000 [-] Log opened.
2018-11-12 08:18:56.742827 [-] Splash version: 3.2
2018-11-12 08:18:56.774236 [-] Qt 5.9.1, PyQt 5.9, WebKit 602.1, sip 4.19.
2018-11-12 08:18:56.774657 [-] Python 3.5.2 (default, Nov 23 2017, 16:37:(
2018-11-12 08:18:56.775068 [-] Open files limit: 1048576
2018-11-12 08:18:56.775331 [-] Can't bump open files limit
2018-11-12 08:18:56.913411 [-] Xvfb is started: ['Xvfb', ':167948583', '-
]
QStandardPaths: XDG_RUNTIME_DIR not set, defaulting to '/tmp/runtime-root'
2018-11-12 08:18:58.785750 [-] proxy profiles support is enabled, proxy pr
2018-11-12 08:18:59.374965 [-] verbosity=1
2018-11-12 08:18:59.375388 [-] slots=50
2018-11-12 08:18:59.375657 [-] argument_cache_max_entries=500
2018-11-12 08:18:59.376684 [-] Web UI: enabled, Lua: enabled (sandbox: ena
2018-11-12 08:18:59.377172 [-] Server listening on 0.0.0.0:8050
2018-11-12 08:18:59.378630 [-] Site starting on 8050
2018-11-12 08:18:59.379114 [-] Starting factory <twisted.web.server.Site (
```

图 25-3 开启 Splash 服务器

在 CMD 窗口或 PyCharm 的 Terminal 窗口下输入指令 scrapy crawl Music，启动并运行项目 music。项目运行完成后，打开 music_db 数据库查看数据表 song 的数据信息，如图 25-4 所示；同时打开 D 盘的 full 文件夹查看歌曲下载信息，如图 25-5 所示。

对象 | song @music_db (MyDb) - 表

开始事务 文本 ▾ 筛选 排序 导入 导出

song_id	song_name	song_ablum	song_interval	song_songmid	song_singer	song_url
1	Chicago Blues Shuffle	Summer Fun V	205	000CGbT93aMkzs	Anthony Proveaux	http://isure.
2	Love Obituary	Cava de Blues:	257	0018f9Nx1ew8As	A Contrablues	http://isure.
3	We Don't Know	Cava de Blues:	222	000r1dWk11zFov	A Contrablues	http://isure.
4	Double Wail	Cava de Blues:	227	002wQ9U53JfJuU	A Contrablues	http://isure.
5	Chances	Cava de Blues:	438	0019Uq1l2gnVML	A Contrablues	http://isure.
6	Freight Train	Cava de Blues:	281	000U8F0n2yLtoC	A Contrablues	http://isure.

图 25-4　数据表 song 的数据信息

› 软件 (D:) › full

名称	标题	参与创作的艺...	唱片集	大小
000aKdL70N...	Cool Jazz Blue	Anthony Prov...	Easy Listening Volu...	2,649 KB
000bvHWp0...	Get You Goin	艾尔文和花栗鼠	艾尔文与花栗鼠	2,720 KB
000CGbT93a...	Chicago Blues...	Anthony Prov...	Summer Fun Volum...	2,446 KB
000D9VWZ3...	Hiding	A Contrablues	Blues a l'Estudi: A ...	3,127 KB

图 25-5　歌曲下载信息

25.5　本 章 小 结

本章介绍了使用 Scrapy 编写爬取 QQ 音乐的程序，通过本章的学习，读者应当掌握以下技能：

1. Scrapy 爬取 QQ 音乐的开发顺序

- setting.py：配置爬虫信息，如请求头、数据库信息、文件保存路径。
- items.py：定义存储数据对象，主要存储歌曲相关信息。
- pipelines.py：数据存储，实现歌曲信息入库和歌曲下载。
- spiders 文件夹（musicSpider.py）：编写爬虫规则。

2. 项目配置

setting.py 是对整个项目的配置，本项目的配置如下：

- Item Pipelines：项目创建时，默认配置 MusicPipeline 类。还需要添加 DownloadMusicPipeline 下载类。该类继承自父类 FilesPipeline，主要实现歌曲下载功能。
- 数据库信息：该配置属于自定义配置信息，变量 MYSQL_CONNECTION 是字符串格式，内容是 SQLAlchemy 连接数据库语句。数据库系统为本地数据库系统，数据库为 music_db。
- 请求头：配置默认的请求头内容，如果项目中发送 HTTP 请求时并没有指定请求头，就默认使用该配置作为请求头。
- 文件保存路径：属于自定义配置信息，变量 FILES_STORE 为字符串格式，内容是系统有效路径，主要用于歌曲下载的保存路径。
- items.py 主要定义歌曲信息寄存的对象，衔接 Spider 和 Item Pipelines，使两者之间的数据交互传递。
- Item Pipelines 主要实现两个功能：歌曲信息入库和歌曲下载。两个功能的数据来源都是 Items 所定义的数据对象。歌曲信息入库主要由 MusicPipeline 实现，歌曲下载由类方法 get_media_requests()和 file_path()共同实现，两者都是从父类 FilesPipeline 继承并重写的。

3. 爬虫规则 Spider

爬虫规则 Spider 共有 6 个类方法，分别说明如下。

- start_requests：歌手字母分类 A-Z，重写 Spider 的 start_requests。
- get_genre_singer：获取每个字母分类下的每页歌手。
- get_singer_songs：获取每一个歌手的信息。
- get_singer_info：获取歌手的每一页歌曲。
- get_song_info：获取每一页的每一首的歌曲信息。
- get_data：每一首歌曲的信息。

从整个 Spider 分析，实现步骤是：全部字母分类歌手列表→当前字母分类歌手列表→当前分类每页的歌手列表→当前分类当前页数的每位歌手→当前歌手的每页歌曲列表→当前歌手的当前歌曲页数的每一首歌曲→获取歌曲信息并下载歌曲。

从实现步骤与第 18 章实现步骤的对比可以发现两者的顺序是相反的，前者是从大到小、从面到点的实现方式；后者是从小到大、从点到面的实现方式。

第 26 章

爬虫的上线部署

26.1　非框架式爬虫部署

当我们完成爬虫的开发后，项目就会进入交付阶段，即将项目的相关文件和程序都一并交付给使用者（也称为甲方）。项目交付涉及到项目部署的问题，对于非框架式的爬虫程序其部署方式如下：

（1）创建可执行程序，使用者只需双击程序即可运行爬虫。

（2）制定任务计划程序，利用计算机的任务管理器来控制爬虫程序的运行时间，无需使用者操作，实现自动化爬虫。

（3）创建服务程序，利用计算机的服务程序来运行爬虫程序，只要计算机没有关机，爬虫每时每刻都在运行。

26.1.1　创建可执行程序

对于爬虫使用者来说，他们并不会开发程序，更不会使用指令去运行程序，尽管他们能使用指令运行程序，但还要为他们的计算机搭建开发环境。若以这样的方式交付项目，使用者是肯定不乐意接受的，因此我们还需要将程序打包成可执行程序。

可执行程序是可在操作系统存储空间中浮动定位的二进制可执行程序，它可以加载到内存中，并由操作系统加载并执行。以 Windows 操作系统为例，它的可执行程序主要以.EXE 为后缀的文件表示，比如常见的 QQ、谷歌浏览器等这类软件。

若将爬虫程序打包成 EXE 文件，前提条件是爬虫程序使用非框架式开发，非框架式开发是指使用 urllib、requests 和 BeautifulSoup 等这类爬虫模块实现的爬虫程序，整个过程不涉及爬虫框架，如 Scrapy 和 PySpider 等。

Python 将 py 文件打包成 EXE 文件需要借助第三方模块实现，目前第三方的打包模块主要有

三种：py2exe、PyInstaller 和 cx_Freeze。本书以 PyInstaller 为例，讲述如何将 py 文件打包成 EXE 文件。

使用 PyInstaller 打包 EXE 之前，需要安装 PyInstaller 模块，安装方式可以使用 pip 指令完成，也可以自行下载 whl 安装包，安装指令如下所示：

```
# pip 在线安装 pyinstaller
pip install pyinstaller
# 从 whl 安装包安装 pyinstaller
pip install PyInstaller-3.4-py2.py3-none-any.whl
```

PyInstaller 成功安装后，我们在 D 盘里创建文件夹 spider 并在文件夹里创建 mySpider.py 文件，然后打开 mySpider.py 文件，编写一个简单的爬虫程序，代码如下：

```
import requests
url = 'https://www.python.org/'
r = requests.get(url)
f = open(r'd:\\spider\\data.txt', 'w')
f.write(r.text)
f.close()
```

上述代码是访问 Python 的官方网页，并将网页内容写入 spider 文件夹的 data.txt 文件。下一步在 D 盘创建 pack 文件夹，打开 CMD 窗口并将 CMD 当前的路径切换到 pack 文件夹所在路径，然后在 CMD 窗口输入程序打包指令 pyinstaller D:\spider\mySpider.py，如图 26-1 所示。

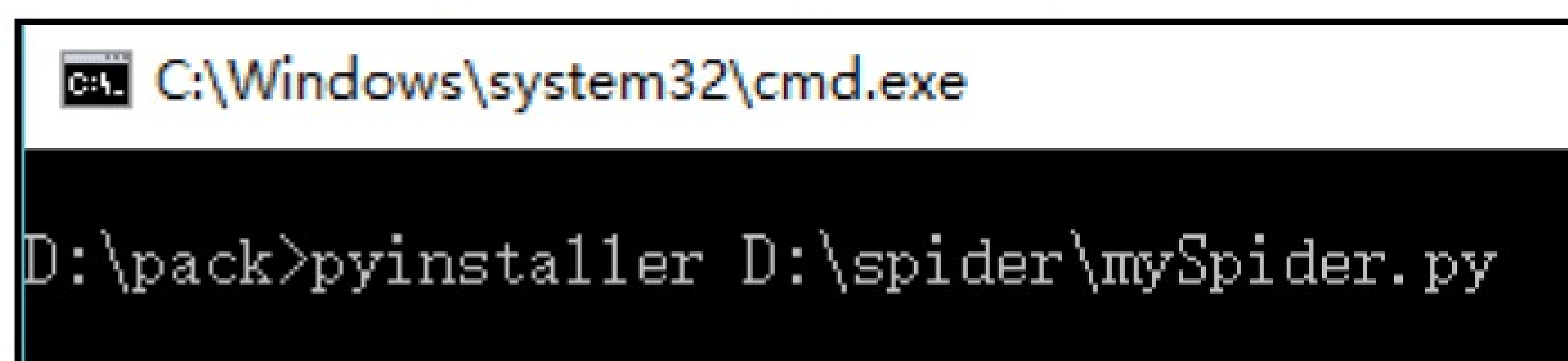


图 26-1　程序打包过程

按回车键运行指令的时候，CMD 窗口会出现相关的运行信息，直到出现“completed successfully”就代表程序已打包成功。由于 CMD 窗口的路径是在 D 盘的 pack 文件夹，因此打包后的 EXE 程序会存放在 pack 文件夹，我们也可以从运行信息里找到 EXE 所在的路径信息，如图 26-2 所示。

```
12352 INFO: Appending archive to EXE D:\pack\build\mySpider\mySpider.exe
12355 INFO: Building EXE from EXE-00.toc completed successfully.
12359 INFO: checking COLLECT
12359 INFO: Building COLLECT because COLLECT-00.toc is non existent
12360 INFO: Building COLLECT COLLECT-00.toc
12560 INFO: Building COLLECT COLLECT-00.toc completed successfully.
```

图 26-2　程序打包成功

打开 EXE 所在的文件夹，发现除了 mySpider.exe 之外，还夹带了很多 pyd 和 dll 文件。当我们双击运行 mySpider.exe 后，程序会出现一个运行窗口，如果程序运行过程中出现错误都会显示在运行窗口上。程序运行完成后，运行窗口会自动关闭，如果窗口没有出现错误信息，说明程序运行成功，如图 26-3 所示。此外，我们也可以验证文件夹 spider 是否出现 data.txt 文件来验证程序是

否运行正常。

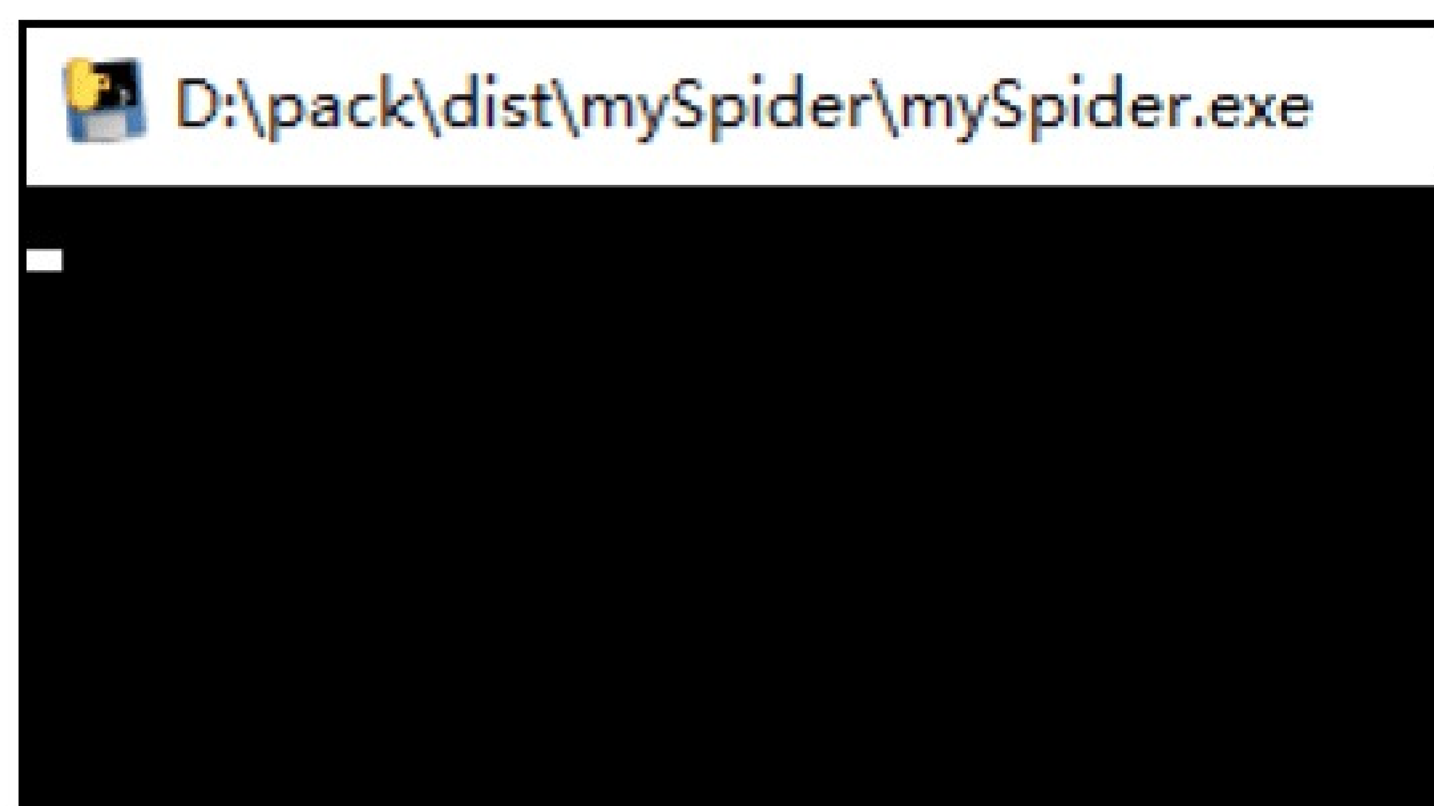


图 26-3　运行窗口

由于 mySpider.exe 所在的文件夹带有很多 pyd 和 dll 文件，如果使用者因操作不当而误删了某个文件，这样会导致 mySpider.exe 无法执行。为了更好地解决这一问题，我们在打包的时候可以加入参数，把依赖文件一并打包到 mySpider.exe 文件里，在 CMD 窗口输入程序打包指令 pyinstaller -F -w D:\spider\mySpider.py，如图 26-4 所示。

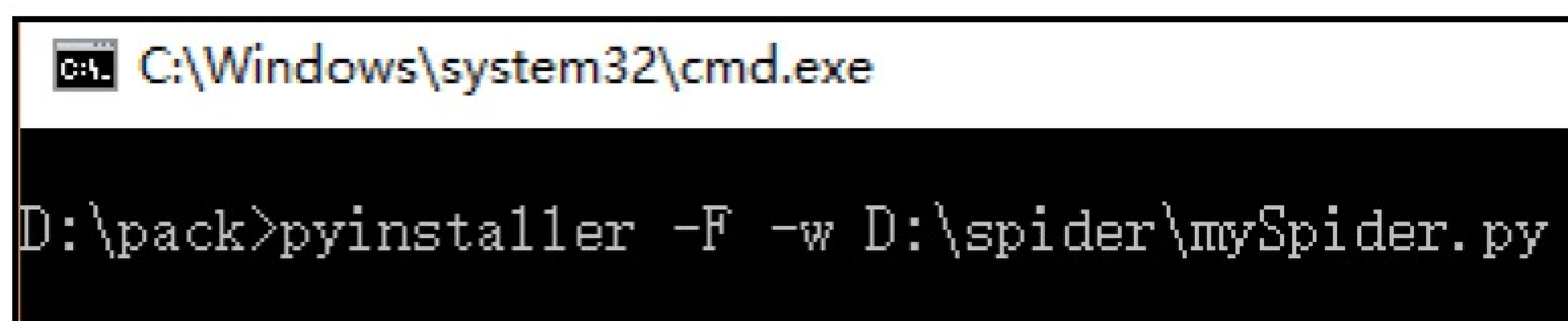


图 26-4　程序打包过程

参数-F 必须为大写，这是将依赖文件一并打包到 mySpider.exe 文件；参数-w 必须为小写，这是隐藏程序控制台，可以美化程序运行，因为 Python 某些模块会出现程序控制台，如 PyQt5 和 Selenium。程序打包成功后，在 D:\pack\dist\路径下找到 mySpider.exe 文件，发现依赖文件已压缩到 EXE 文件，mySpider.exe 的大小从 2MB 变为 7MB，如图 26-5 所示。



图 26-5　mySpider.exe 文件

26.1.2　制定任务计划程序

虽然我们已将爬虫程序打包成可执行程序给使用者使用，但对于一些要求比较高的使用者来说，每次运行爬虫都要自己双击运行程序，显得不够人性化，那么是否不用双击就能自动运行爬虫程序呢？本节，我们将讲述如何将可执行程序实现自动化运行。

在操作系统中，有一个系统功能叫做定时任务，不同的操作系统，定时任务都有不同的操作方式。以 Windows10 为例，定时任务也称为任务计划程序，打开任务计划程序的方式有多种，这

里在计算机的“控制面板”里找到并单击“管理工具”图标；在管理工具的窗口中可以看到“任务计划程序”图标，单击后即可打开任务计划程序窗口，如图 26-6 所示。

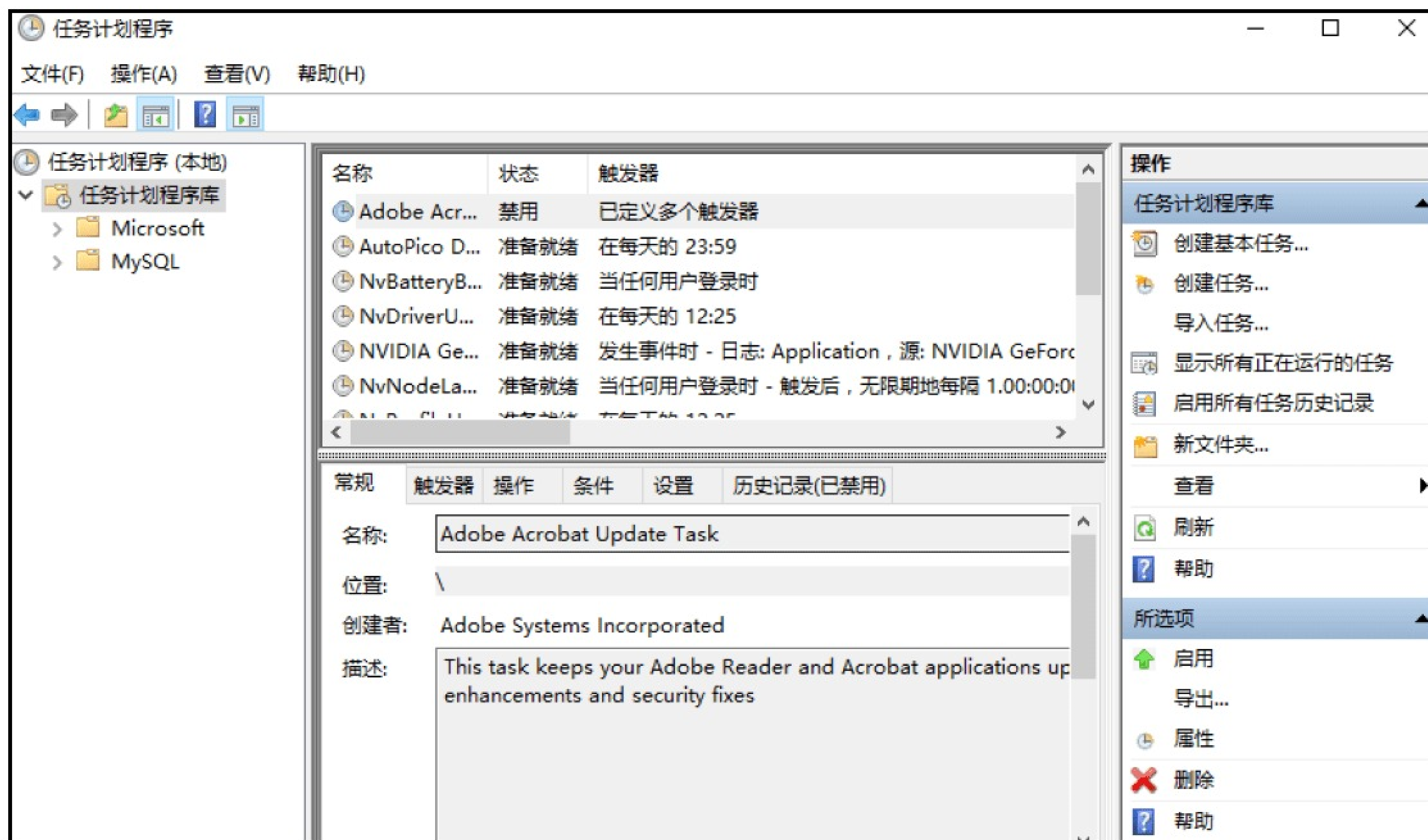


图 26-6 打开任务计划程序

从任务计划程序窗口看到，窗口正上方是任务计划列表，每个任务计划都有常规、触发器、操作、条件、设置和历史记录等基本功能，各个功能说明如下。

- 常规：设置当前任务计划的名称、位置、创建者和任务描述等基本信息。
- 触发器：设置任务的触发条件，比如任务的运行时间、运行间隔、运行条件等信息。
- 操作：指定当前任务所运行的程序，这是将程序与任务进行绑定。
- 条件：与触发器一起作为任务运行的条件，如设置计算机的状态、电源状态和网络状态来决定是否执行任务。
- 设置：根据任务运行期间的状态进行设置，如设置运行超时、运行失败等异常操作。
- 历史记录：负责记录运行信息，Windows10 系统已被禁用。

大致了解任务计划程序的基本功能后，接下来是创建任务计划，单击图 26-6 右侧的创建任务就会出现创建任务窗口，该窗口出现 5 个功能选项卡，分别用来设置任务的基本功能，如图 26-7 所示。

在“常规”选项卡里，一般设置任务名称和描述即可，其他设置使用默认即可，我们将任务名称设为 mySpider，描述内容为“run mySpider.exe”即可完成常规选项卡的设置。然后单击“触发器”选项卡，单击“新建”按钮，创建新的触发器，如图 26-8 所示。

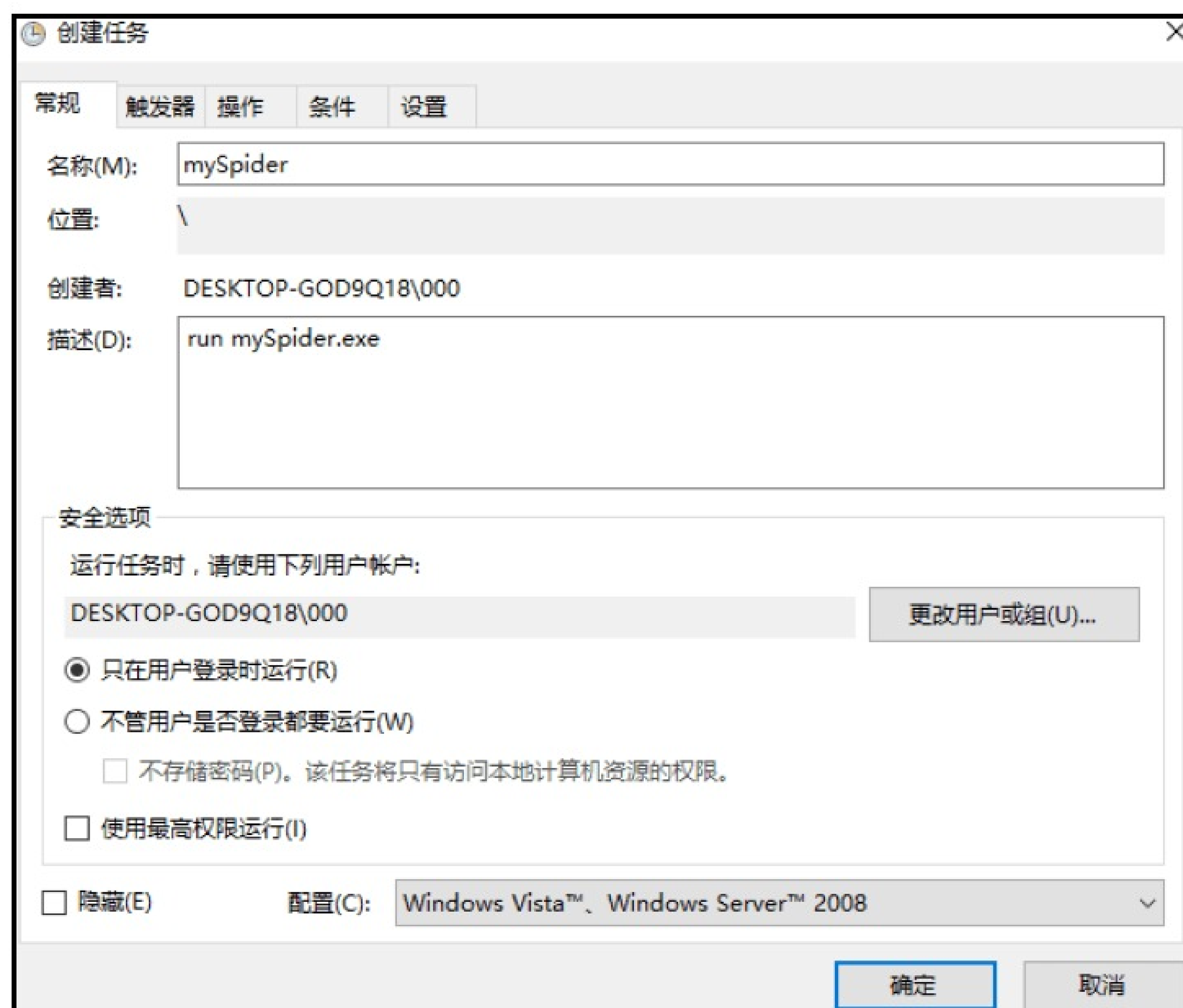


图 26-7　创建任务窗口

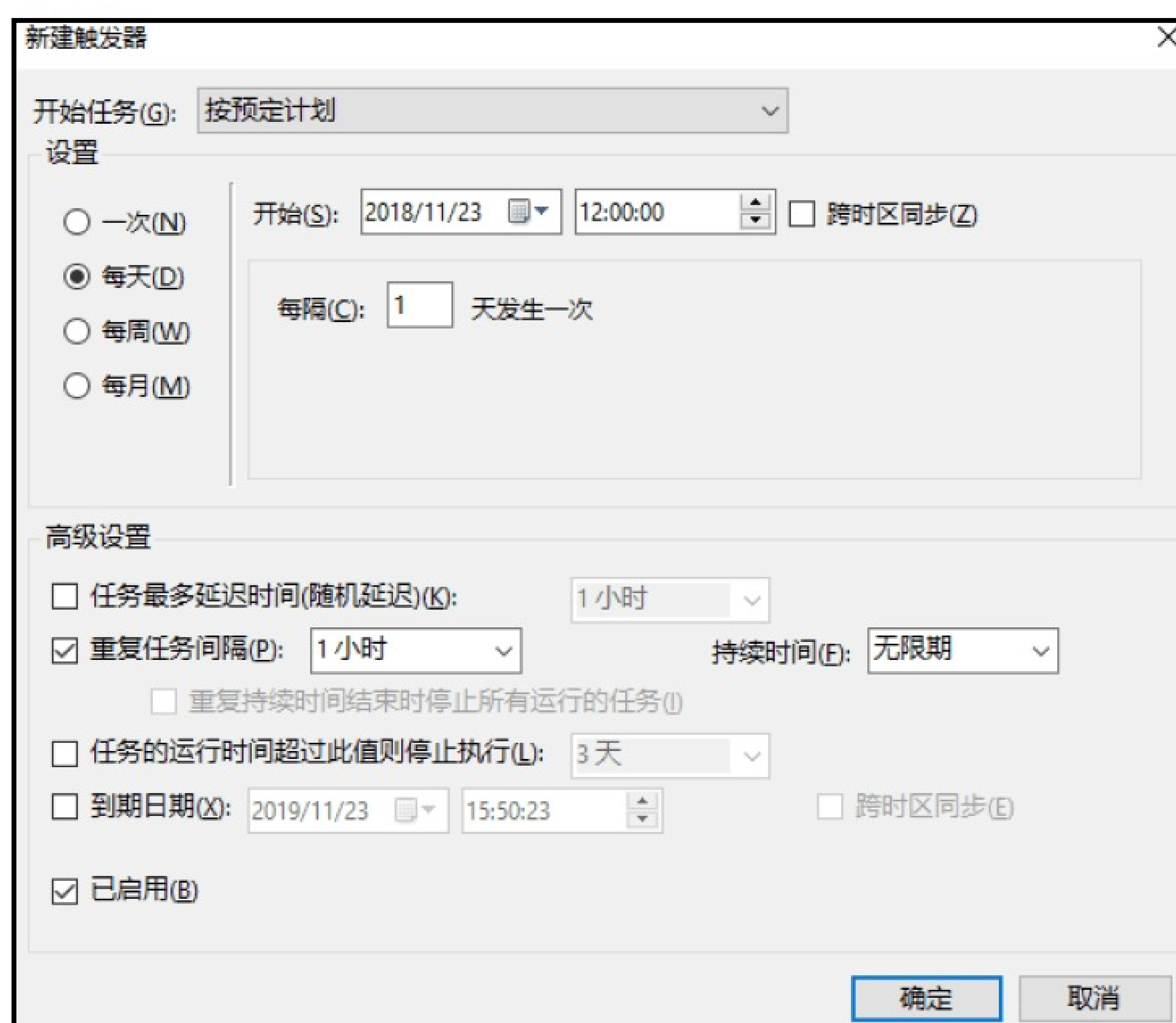


图 26-8　新建触发器

从新建触发器的窗口看到，开始任务是一个文本下拉框，默认值是按预定计划，下拉框还有其他触发条件，如用户登录 Windows 时触发、启动 Windows 时触发和发生事件时触发等触发条件，不同的触发条件有不同的设置。

以按预定计划为例，将触发器设为每天中午 12 点开始运行，每隔 1 小时就运行一次，持续时间为无限期，该设置说明：只要电脑没有关机，该任务就会无限期运行，运行间隔为 1 小时；如果电脑关机重启了，当电脑时间到了中午 12 点就会无限期运行。

触发器创建后，还可以继续添加新的触发器或者编辑（删除）已有的触发器，一个任务计划里可以有多个触发器，以便满足多方面的需求，如图 26-9 所示。

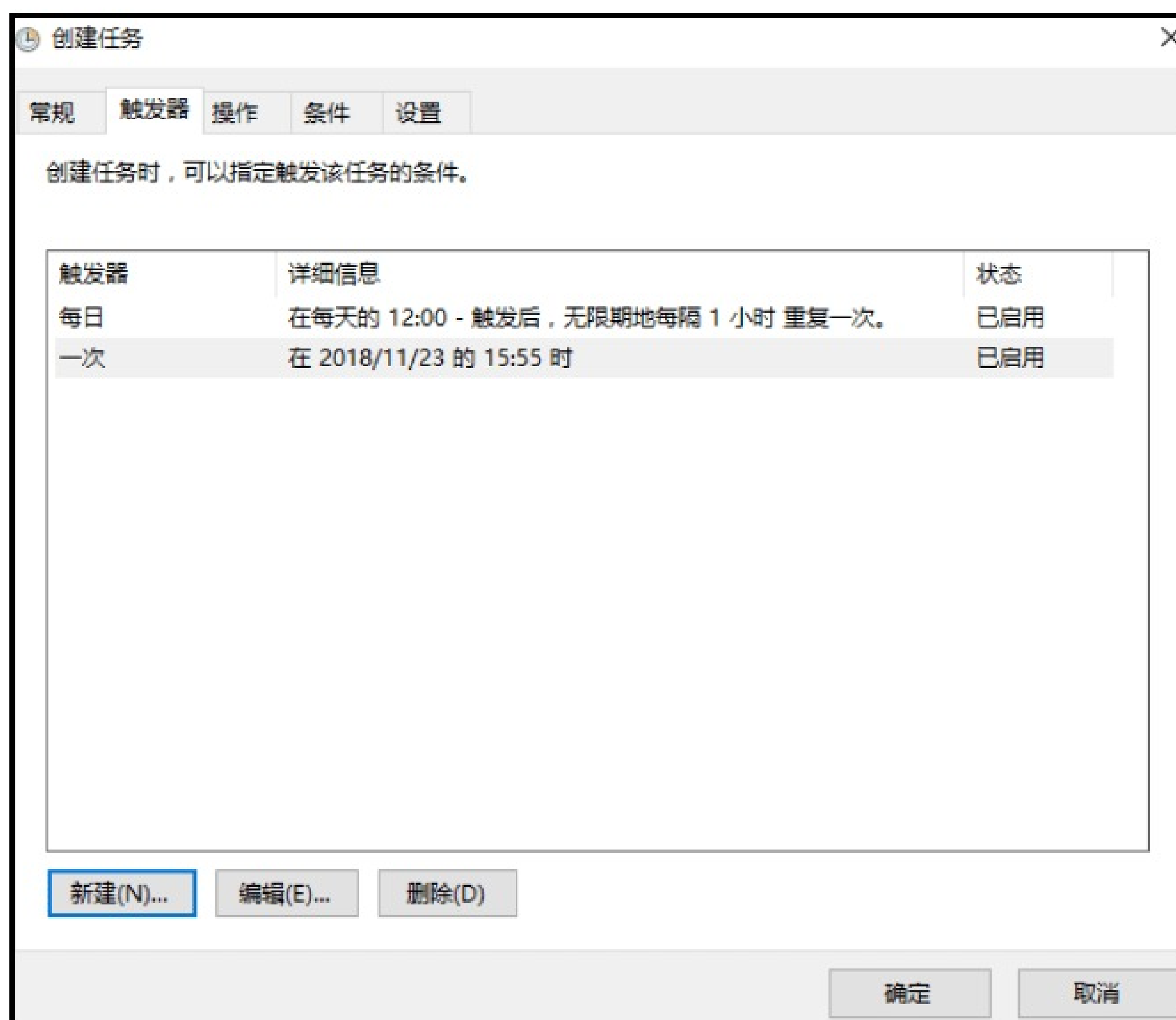


图 26-9　触发器选项卡

下一步单击“操作”选项卡，将爬虫程序 mySpider.exe 与计划任务进行绑定，单击“新建”按钮，创建新的操作。在新建操作的窗口里单击浏览按钮并将爬虫程序 mySpider.exe 选中，如图 26-10 所示。

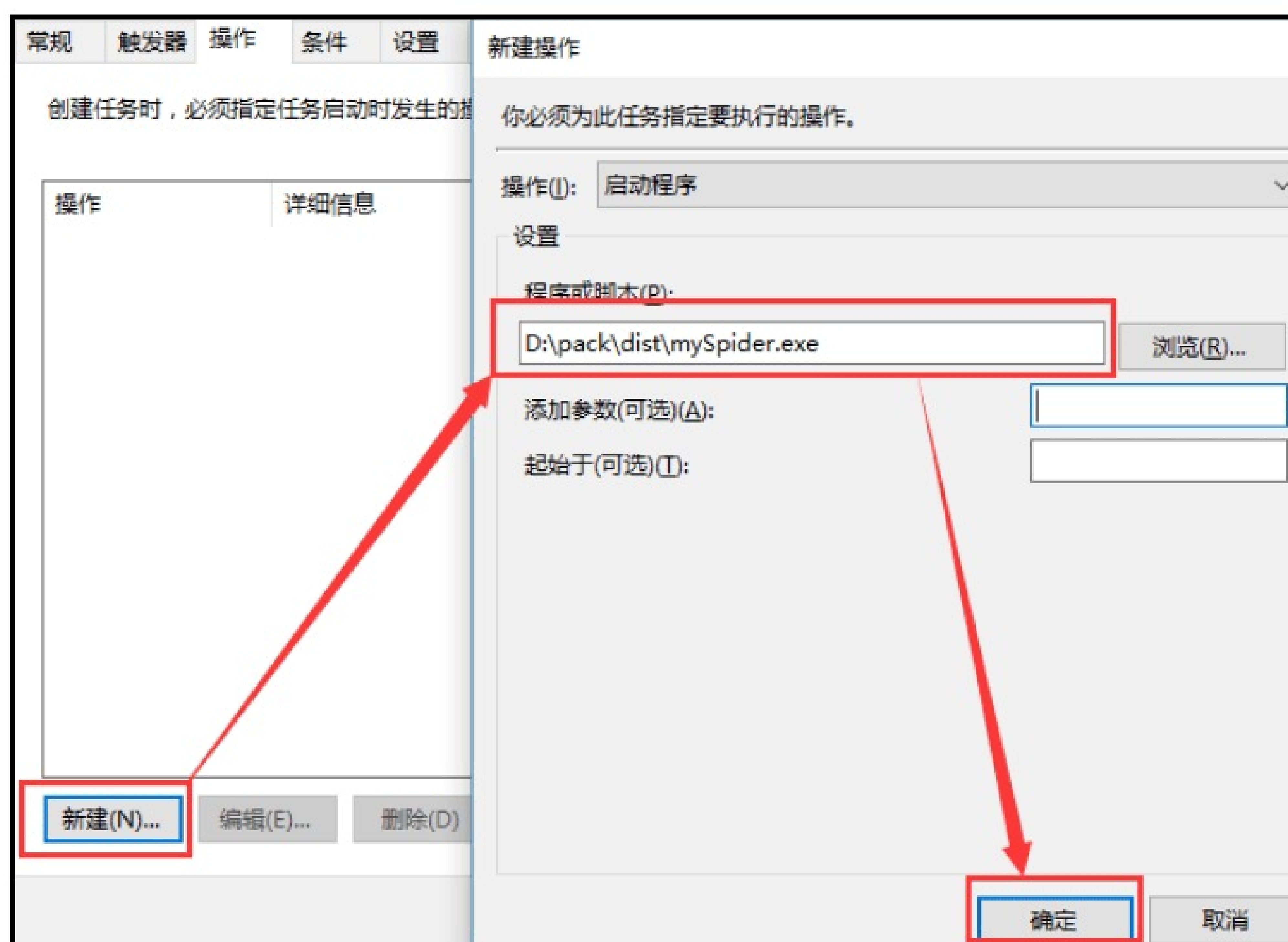


图 26-10　新建操作

操作创建后，还可以继续添加新的操作或者编辑（删除）已有的操作，一个任务计划里也可以支持多个操作同时存在。对于条件和设置选项卡，一般无须设置，使用默认值即可满足大部分的需求，因此本书就不再详细讲述。

任务创建成功后，在任务计划程序窗口里找到新创建的任务 mySpider，若想对已创建的任务进行修改，只需双击任务即可进入修改窗口，如图 26-11 所示。

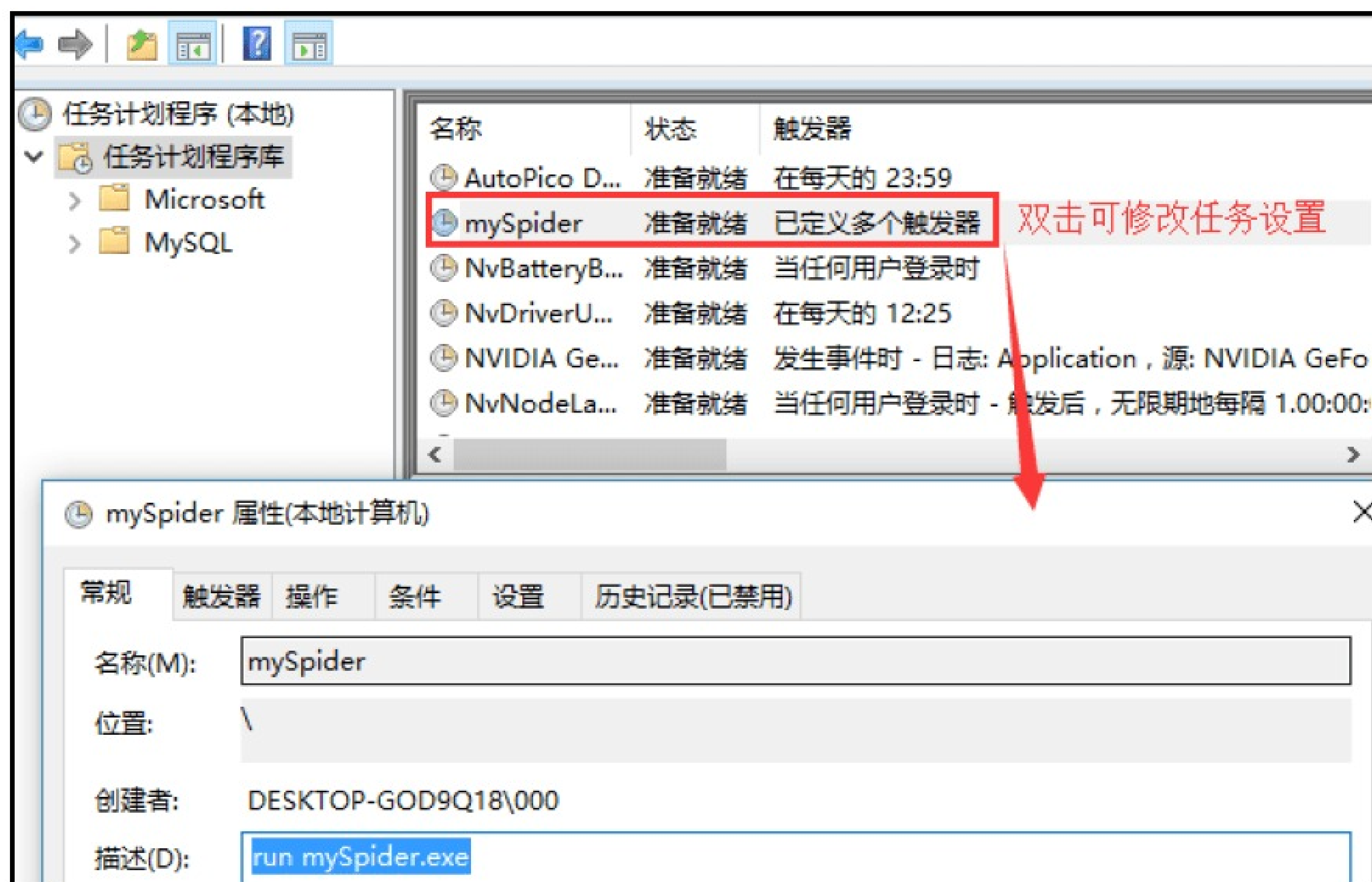


图 26-11　修改已有任务计划

26.1.3　创建服务程序

对于一些更为刁钻的使用者来说，任务计划程序依然不能满足他们的使用需求，他们希望爬虫程序每时每刻都在运行或者处于待命的状态，且程序在运行的时候又不能出现程序运行窗口，而且只希望通过配置文件来控制爬虫的运行和待命状态。

要使爬虫每时每刻处于活动状态，可以将爬虫程序设为一个死循环，每次循环的等待时间设为 10 秒；想要爬虫程序长期运行又不能出现程序运行窗口，只能将爬虫程序以服务程序的形式运行。首先我们将爬虫程序 mySpider.py 的功能进行修改，修改后的代码如下：

```
import requests
import time
while 1:
    # 读取配置文件的任务内容
    taskFile = open(r'd:\\spider\\task.txt')
    task = taskFile.read()
    taskFile.close()
    for i, url in enumerate(task.split(',')):
        if url:
            r = requests.get(url)
            path = r'd:\\spider\\%s.txt' %(str(i))
            f = open(path, 'w',encoding='utf-8')
            f.write(r.text)
            f.close()
    # 清空配置文件的任务内容
    taskFile=open(r'd:\\spider\\task.txt','w')
    taskFile.write('')
```

```
        taskFile.close()
        time.sleep(10)
```

上述代码将爬虫程序设为一个死循环，每次循环都会读取配置文件 task.txt 的内容，从文件内容提取每个 URL 地址并对其实现 HTTP 请求，将响应内容写入新的 txt 文件，最后清空配置文件内容，防止下次循环重复执行。

将修改后的 mySpider.py 文件进行打包处理，生成 mySpider.exe 文件。下一步是把 EXE 文件生成服务程序，这个实现过程需要使用辅助工具 NSSM，在浏览器中打开 NSSM 网站（http://www.nssm.cc/download）下载 NSSM 工具，如图 26-12 所示。

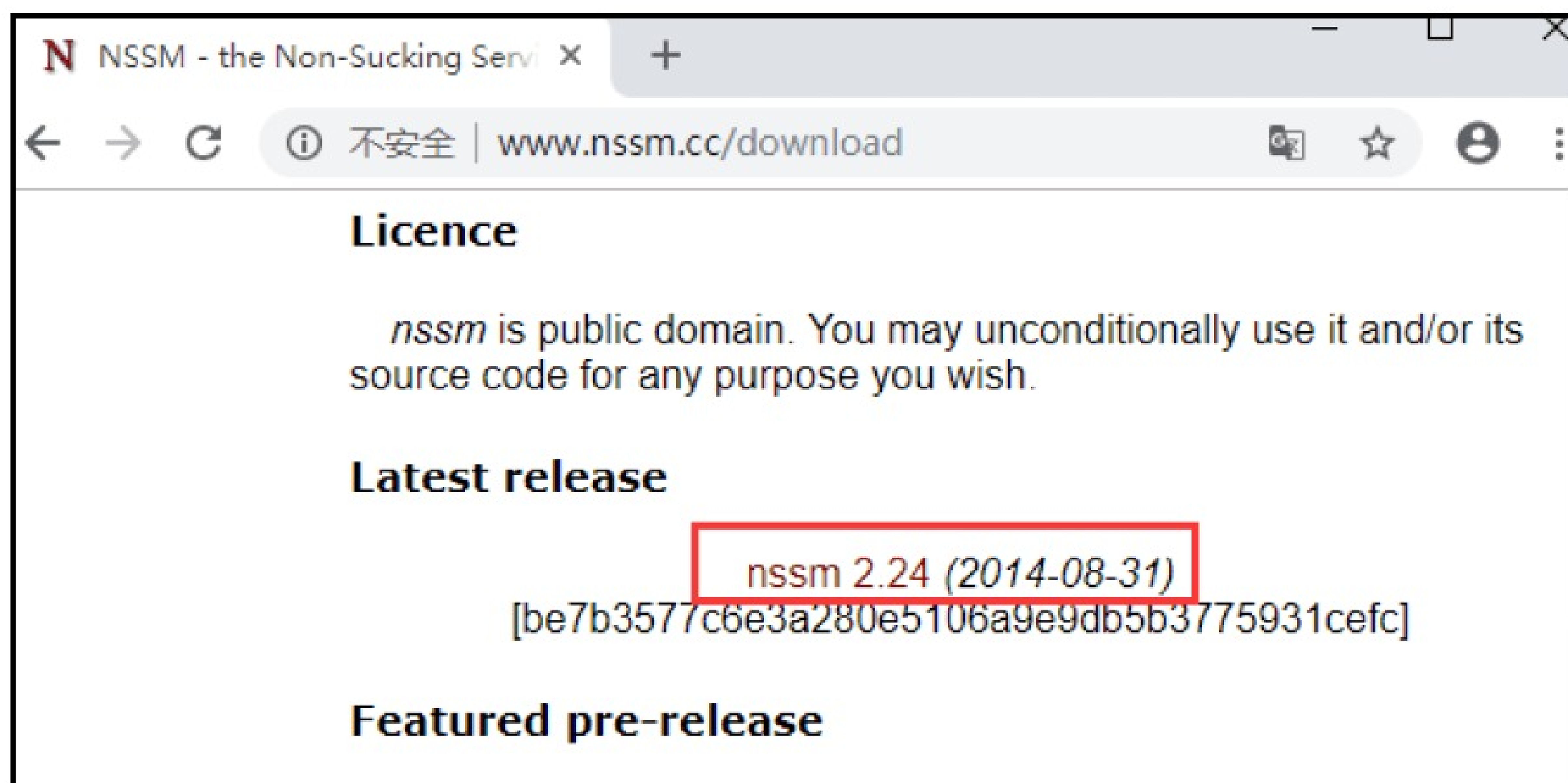


图 26-12　下载 NSSM

将下载后的 NSSM 压缩包进行解压处理，然后根据计算机的位数选择相应的文件夹。以计算机的 64 位为例，打开 CMD 窗口并将路径切换到 NSSM 的 win64 文件夹，输入 NSSM 指令（nssm install）运行 NSSM，如图 26-13 所示。

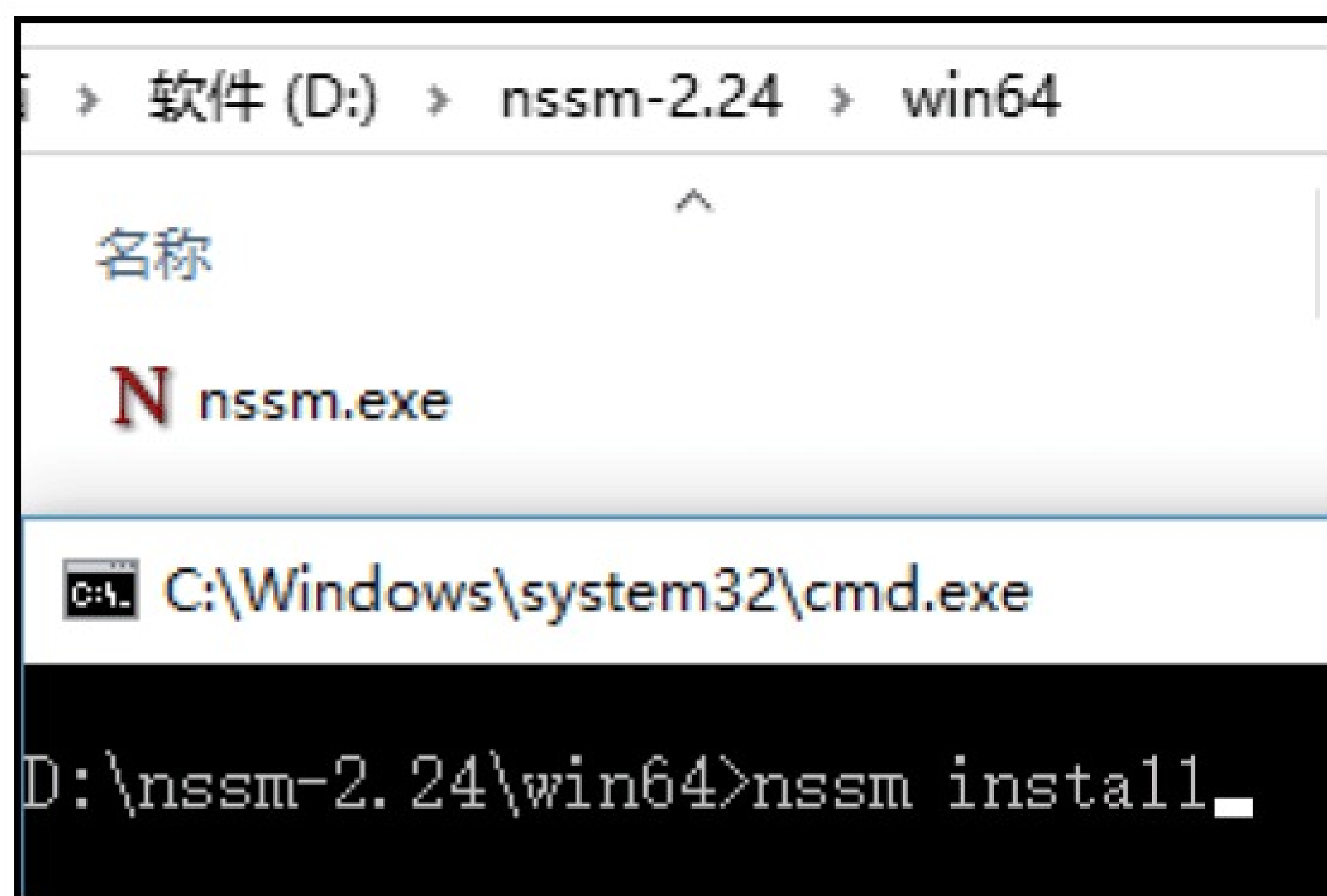


图 26-13　运行 NSSM 工具

NSSM 只能通过 CMD 窗口运行，并且 CMD 窗口的路径必须为 nssm.exe 所在的路径。如果直接双击 nssm.exe 只会出现 NSSM 的指令信息，读者可以在 CMD 窗口输入指令信息来使用 NSSM 功能。现在需要使用 NSSM 安装服务程序，输入 nssm install 指令后会出现 NSSM service installer 窗口，如图 26-14 所示。

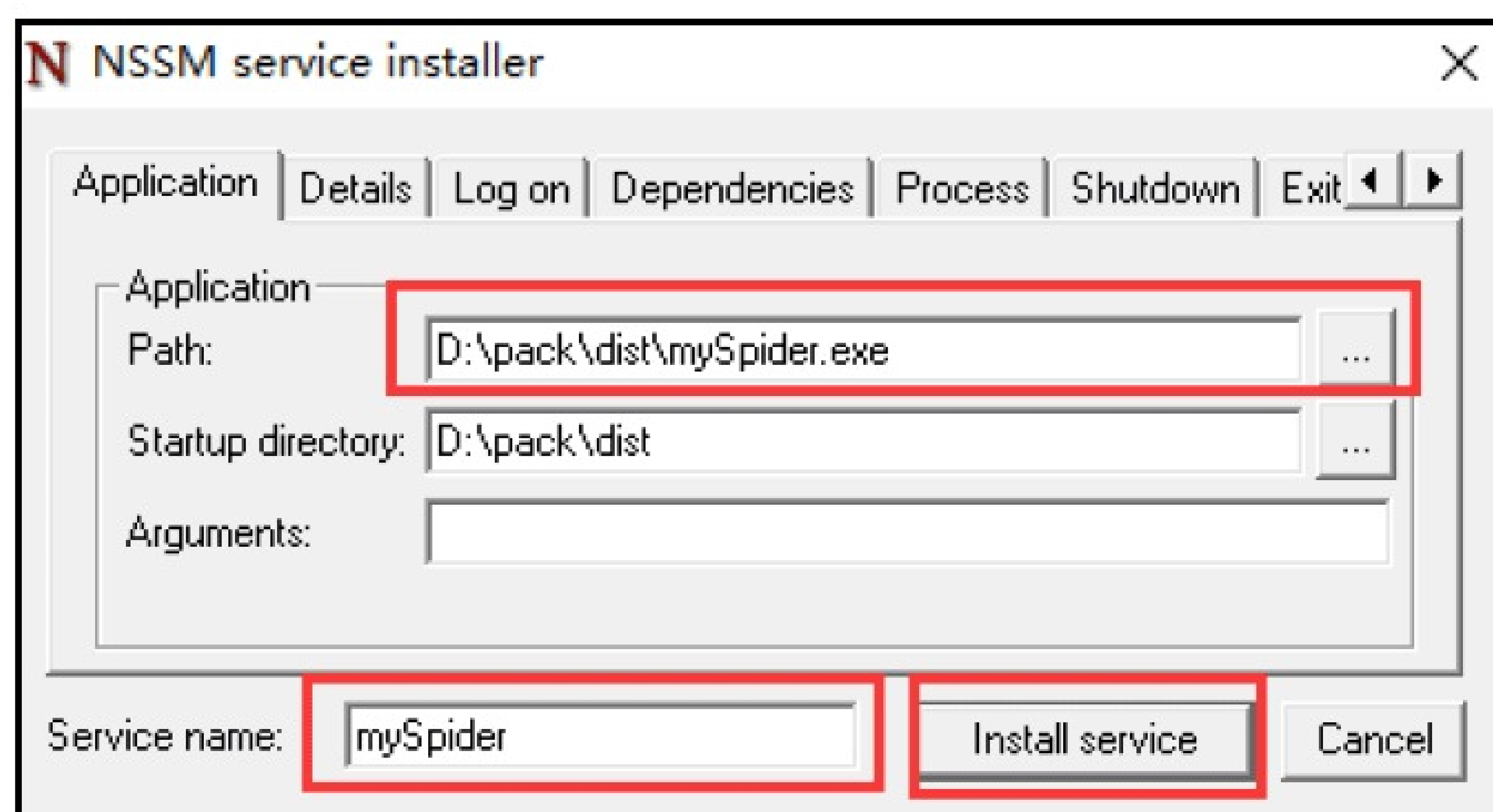


图 26-14　NSSM service installer 窗口

在 NSSM service installer 窗口里，Path 文件框选择已打包好的 mySpider.exe 文件，Service Name 是对服务程序进行命名，我们将其命名为 mySpider，最后单击 Install service 按钮即可安装服务程序。

服务程序安装成功后，在 Windows10 桌面，右键单击桌面上的“此电脑”图标，在弹出的菜单中选择“管理”菜单项；在打开的计算机管理窗口中，单击左侧的“服务和应用程序/服务”菜单项；在右侧的窗口中就会打开服务程序；在服务程序列表中找到 mySpider 服务并单击“启动此服务”，如图 26-15 所示。

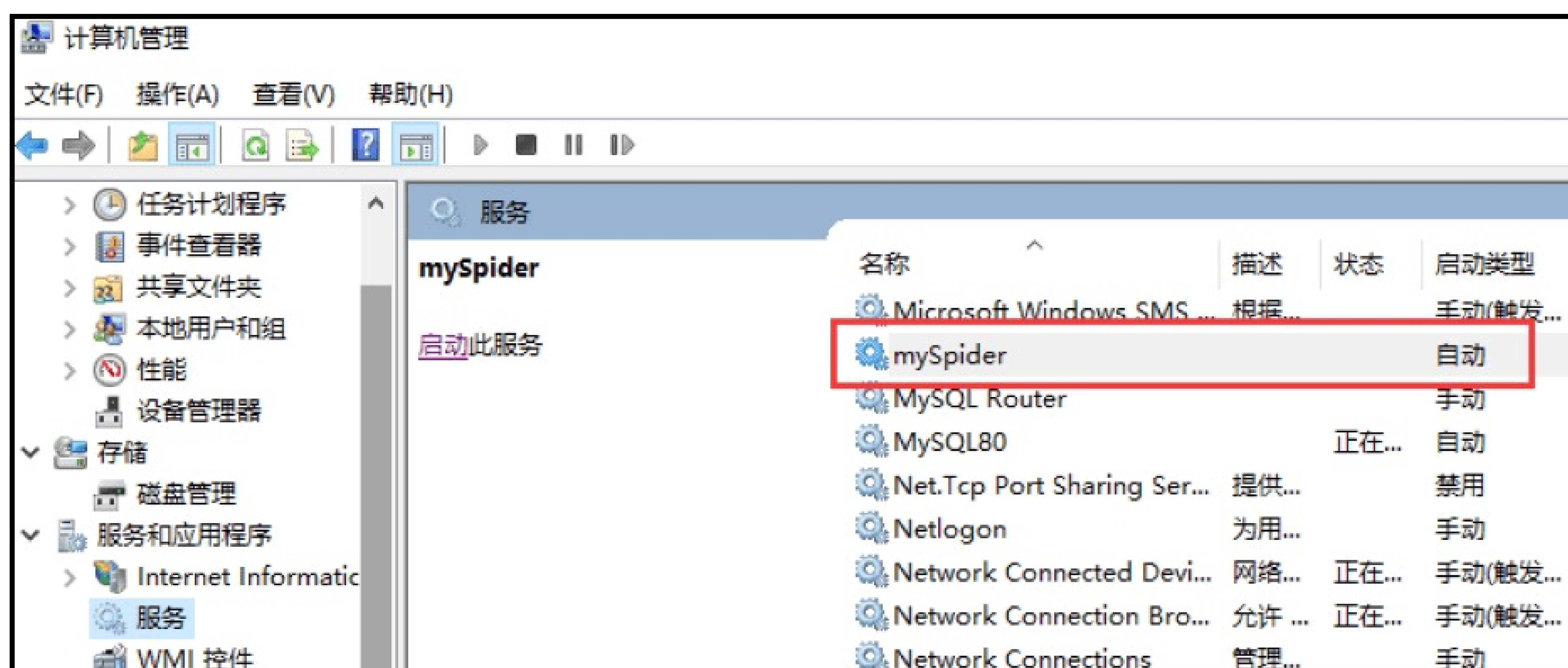


图 26-15　启动 mySpider 服务

服务程序启动后，爬虫程序每隔 10 秒就会运行一次，如果 D:\spider\task.txt 的文件内容不为空（文件内容如图 26-16 所示），爬虫程序就会爬取数据并且清空文件内容；如果文件内容为空，爬虫就不做任何操作，可认为是处于一种待命状态。



图 26-16　文件内容

服务程序开启后，只要计算机处于开机状态，服务就会一直处于运行状态，如果不想使用服

务，可在服务程序列表中停止服务。此外，还可以将服务从服务程序列表删除。我们右键单击“CMD”图标，在弹出菜单中选择“管理员身份运行”菜单项，在弹出的 CMD 窗口上输入删除指令 sc delete mySpider 即可删除服务程序 mySpider，如图 26-17 所示。

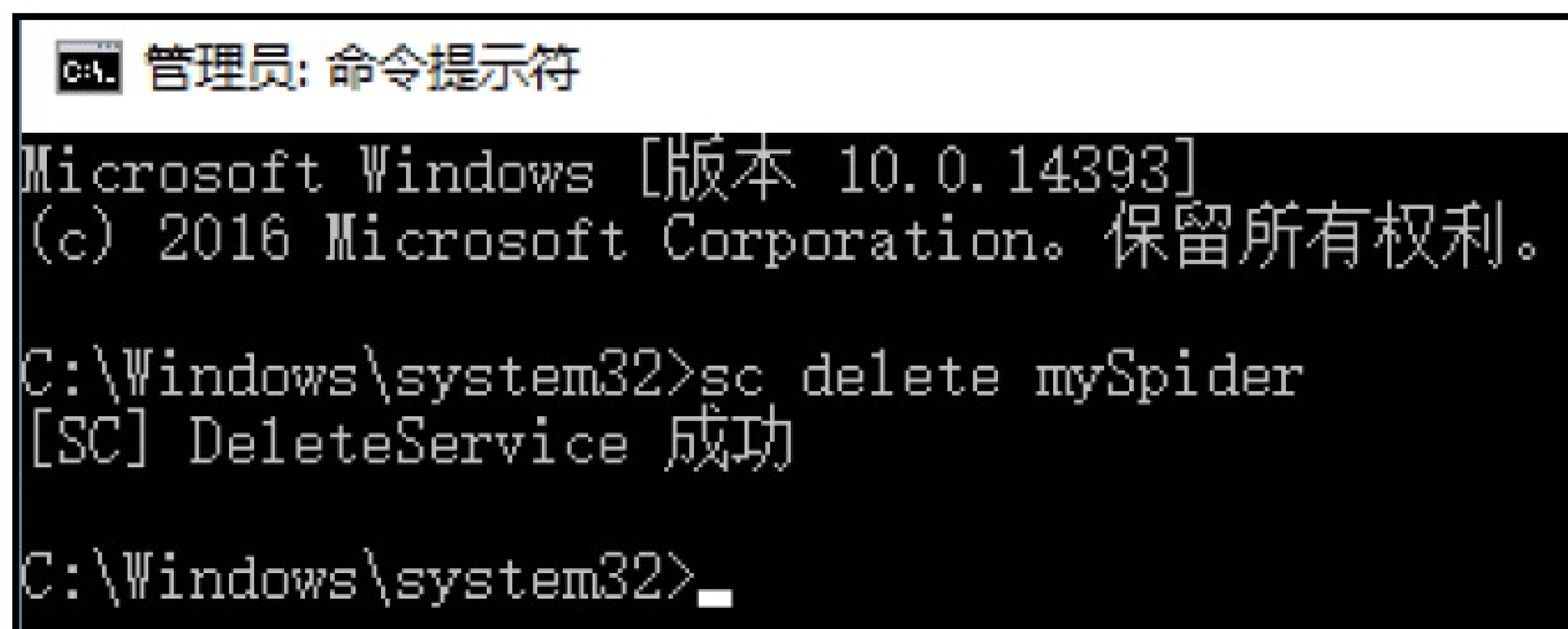


图 26-17 删除服务程序 mySpider

创建服务程序除了使用 NSSM 工具之外，还可以使用 Python 的 pywin32 模块实现，利用 pywin32 将爬虫程序进行包装处理，然后将爬虫程序生成 EXE 文件，最后把 EXE 文件安装到电脑上即可生成服务程序，具体的实现过程本书不再详细讲述。

26.2 框架式爬虫部署

框架式爬虫是指使用 Scrapy 或 PySpider 等爬虫框架开发的爬虫程序，此类爬虫程序有自身的运行方式。对于这种爬虫的部署，只能通过特定的工具来实施，我们以 Scrapy 框架为例，部署工具主要用 Scrapyd 和 Gerapy。

26.2.1 Scrapyd 部署爬虫服务

Scrapyd 是一款用于管理 Scrapy 爬虫的部署和运行的服务，由 Scrapy 的开发者开发，它是通过 API 接口来部署或者控制 Scrapy 项目，可以管理多个项目，并且每个项目还可以上传多个版本，但只有最新的版本会被使用。Scrapyd 是开源软件，代码托管于 Github（https://github.com/scrapy/scrapyd）。

使用 Scrapyd 之前，需要在 Python 的开发环境下安装 Scrapyd 和 Scrapyd-Client 模块，安装方式可以使用 pip 指令完成，安装指令如下：

```
# pip 在线安装 Scrapyd
pip install scrapyd
# pip 在线安装 Scrapyd-Client
pip install scrapyd-client
```

在使用 Scrapyd 管理 Scrapy 项目之前，还要对 Scrapy 项目进行打包处理，打包过程可以选择 Scrapyd-Client 模块实现或者使用 Scrapyd 的 API 接口。本节以项目 douban 为例，讲述如何使用 Scrapyd-Client 模块实现项目打包。

Scrapyd-Client 模块安装成功后，还需要在 Python 安装目录的 Scripts 文件夹创建脚本文件，

脚本文件命名为 scrapyd-deploy.bat，如图 26-18 所示。

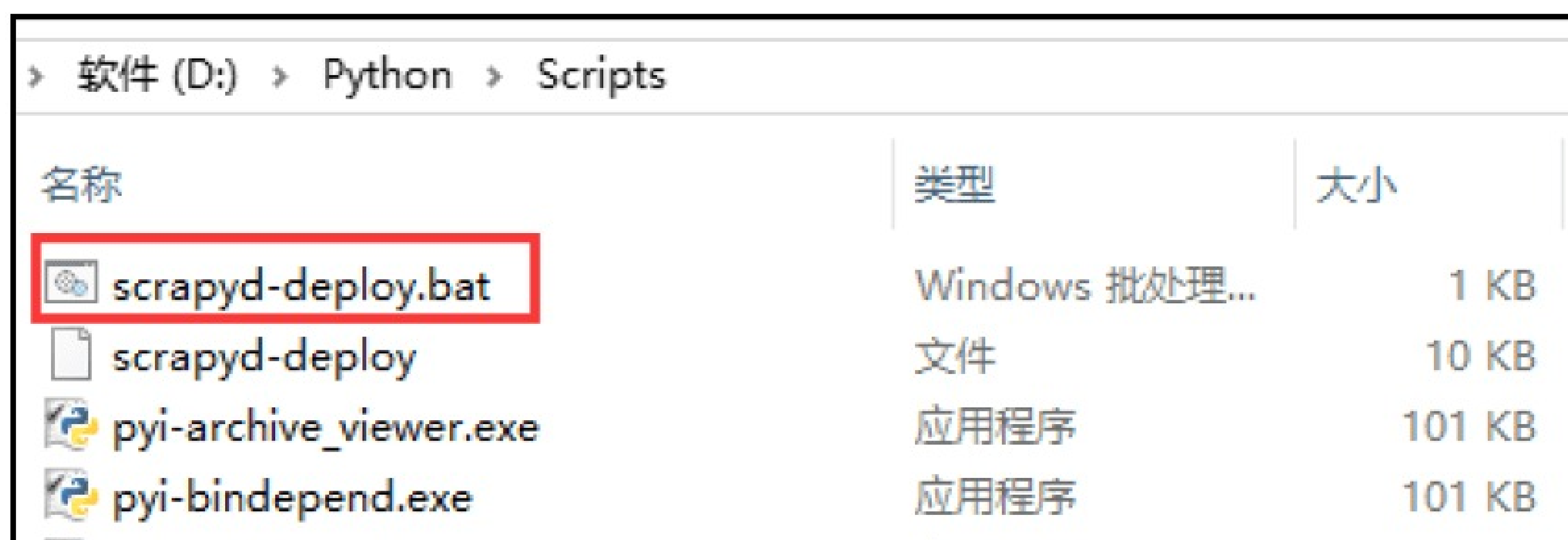


图 26-18　创建脚本文件

打开脚本文件 scrapyd-deploy.bat，在文件里编写脚本代码，其作用是通过 Python 指令运行 Scripts 文件夹的 scrapyd-deploy 文件，由该文件去执行项目打包过程，代码如下：

```
@echo off
"D:\Python\python.exe" "D:\Python\Scripts\scrapyd-deploy"
%1 %2 %3 %4 %5 %6 %7 %8 %9
```

上述代码的“python.exe”是 Python 安装目录下的 Python 解释器；“scrapyd-deploy”是 Scripts 文件夹的 scrapyd-deploy 文件。读者可根据实际开发环境进行配置，如果不编写脚本文件，则在 CMD 窗口下无法使用指令打包 Scrapy 项目，如图 26-19 所示。

```
D:\douban>scrapyd-deploy
'scrapyd-deploy' 不是内部或外部命令，也不是可运行的程序
或批处理文件。
```

图 26-19　打包指令

脚本文件创建后，下一步是修改项目文件 scrapy.cfg 的配置内容，配置属性 url 代表 Scrapyd 的 IP 地址。假如在一个局域网内，有多台计算机需要部署不同的 Scrapy 项目，而且每台计算机都会装上 Scrapyd 服务，那么配置属性 url 是把当前项目部署到指定的 Scrapyd。配置内容如下：

```
# 一个项目配置单一主机
[deploy]
url = http://localhost:6800/
project = douban

# 一个项目配置多台主机
[deploy:douban1]
url = http://localhost:6800/
project = douban

[deploy:douban2]
url = http://139.217.26.30:6800/
project = douban
```

如果一个项目只部署在一台计算机上，只需定义一个 deploy 属性即可；如果一个项目部署在多台计算机上，那么可定义多个 deploy 属性，并在 deploy 属性后添加冒号和名称，如“deploy:douban1”。

上述方法只是为项目打包做准备工作，接下来是执行打包过程。在 D 盘下创建文件夹 deployScrapy，打开 CMD 窗口并将路径切换到 deployScrapy 的路径，输入 Scrapyd 服务开启指令 scrapyd，如图 26-20 所示。

```
D:\deployScrapy>scrapyd
2018-11-29T14:29:51+0800 [-] Loading d:\python\lib
2018-11-29T14:29:52+0800 [-] Scrapyd web console a
2018-11-29T14:29:52+0800 [-] Loaded.
2018-11-29T14:29:52+0800 [twisted.application.app.
.
2018-11-29T14:29:52+0800 [twisted.application.app.
actor.
2018-11-29T14:29:52+0800 [-] Site starting on 6800
2018-11-29T14:29:52+0800 [twisted.web.server.Site#
A1F828>
2018-11-29T14:29:52+0800 [Launcher] Scrapyd 1.2.0
```

图 26-20　开发 Scrapyd 服务

Scrapyd 服务开启后，再次打开一个新的 CMD 窗口，将路径切换到项目 douban，也就是项目 scrapy.cfg 文件所在路径。如果项目只部署在一台主机上，打包指令为 scrapyd-deploy，如图 26-21 所示。

```
D:\douban>scrapyd-deploy
Packing version 1543463867
Deploying to project "douban" in http://localhost:6800/addversion.json
Server response (200):
{"node_name": "DESKTOP-GOD9Q18", "status": "ok", "project": "douban", "
```

图 26-21　单点部署

如果项目部署在多台主机上，就要多次输入打包指令，比如 deploy 属性设有 douban1 和 douban2，输入两次的打包指令分别为 scrapyd-deploy douban1 和 scrapyd-deploy douban2。由于 deploy 属性为 douban2 的 url 属性是不存在的 IP 地址，因此在打包过程中会出现无法连接的错误信息，如图 26-22 所示。

```
D:\douban>scrapyd-deploy douban1
Packing version 1543472565
Deploying to project "douban" in http://localhost:6800/addversion.json
Server response (200):
{"node_name": "DESKTOP-GOD9Q18", "status": "ok", "project": "douban", "


D:\douban>scrapyd-deploy douban2
Packing version 1543472575
Deploying to project "douban" in http://139.217.26.30:6800/addversion.j
Deploy failed: <urlopen error [WinError 10061] 由于目标计算机积极拒绝,
```

图 26-22　多点部署

项目打包成功后，打开并查看项目 douban 和文件夹 deployScrapy 的文件信息，发现两者都新

增了相关文件和文件夹，如图 26-23 所示。

名称	类型	大小
dbs	文件夹	
eggs	文件夹	
logs	文件夹	

名称	类型	大小
.idea	文件夹	
build	文件夹	
douban	文件夹	
project.egg-info	文件夹	
scrapy.cfg	CFG 文件	1 KB
setup.py	Python File	1 KB

图 26-23　文件信息

在文件夹 deployScrapy 里共有三个文件夹，分别为 dbs、eggs 和 logs，三者存储 Scrapy 项目的数据信息，说明如下：

- dbs 是为已部署的 Scrapy 项目生成相应的数据库文件，Scrapyd 每部署一个项目都会生成一个文件。
- eggs 是 Scrapy 项目的版本信息，如果多次打包同一个项目，则会生成多个版本信息，每次 Scrapyd 运行项目都会选择最新版本。
- logs 是记录 Scrapy 项目的运行信息，每次 Scrapyd 运行项目都会生成一个 txt 文件，记录项目的运行情况。

调用 Scrapyd 的 API 接口来运行 Scrapy 项目，必须保证已开启 Scrapyd 服务并且 Scrapy 项目已做打包处理。调用 API 接口可以使用 Requests 模块，实现过程与编写爬虫是同一个原理。以运行 Scrapy 爬虫和删除爬虫为例，调用方法如下：

```
import requests
# 运行 Scrapyd 的 Scrapy
url = 'http://localhost:6800/schedule.json'
data = {
    # 参数 project 是 scrapy.cfg 的属性 project
    'project': 'douban',
    # 参数 spider 是项目 spider 的 name 属性
    'spider': 'Movie'
}
r = requests.post(url, data=data)
print(r.json())
```

```
# 删除 Scrapyd 的 Scrapy
url = 'http://localhost:6800/delproject.json'
data = {
    # 参数 project 是 scrapy.cfg 的属性 project
    'project': 'douban',
}
r = requests.post(url, data=data)
print(r.json())
```

调用 Scrapyd 的 API 接口只需对请求地址发送指定的 HTTP 请求以及设置指定的请求参数即可，Scrapyd 的官方文档已对每个 API 接口做了详细介绍（https://scrapyd.readthedocs.io/en/latest/api.html），本书就不再重复讲述。

项目部署成功后，如果要对项目的功能进行修改，每次修改后都要重新打包项目，在 Scrapyd 的 eggs 文件夹里生成最新的版本信息，否则 Scrapyd 运行项目的时候，项目还是会以修改前的方式运行。

此外，我们还可以查看 Scrapyd 的配置信息，在 Python 的安装目录下找到 Scrapyd 模块包，在模块包下找到 default_scrapyd.conf 文件（Lib\site-packages\scrapyd\default_scrapyd.conf），使用记事本打开即可查看 Scrapyd 的配置信息，如图 26-24 所示。

```
[scrapyd]
eggs_dir    = eggs
logs_dir    = logs
items_dir   =
jobs_to_keep = 5
dbs_dir     = dbs
max_proc    = 0
max_proc_per_cpu = 4
finished_to_keep = 100
poll_interval = 5.0
bind_address = 127.0.0.1
http_port   = 6800
debug       = off
runner      = scrapyd.runner
application = scrapyd.app.application
launcher    = scrapyd.launcher.Launcher
webroot     = scrapyd.website.Root

[services]
schedule.json     = scrapyd.webservice.Schedule
cancel.json       = scrapyd.webservice.Cancel
addversion.json   = scrapyd.webservice.AddVersion
listprojects.json = scrapyd.webservice.ListProjects
listversions.json = scrapyd.webservice.ListVersions
listspiders.json  = scrapyd.webservice.ListSpiders
delproject.json   = scrapyd.webservice.DeleteProject
delversion.json   = scrapyd.webservice.DeleteVersion
listjobs.json     = scrapyd.webservice.ListJobs
daemonstatus.json = scrapyd.webservice.DaemonStatus
```

图 26-24　Scrapyd 的配置信息

综上所述，Scrapyd 管理和部署 Scrapy 项目的操作方法如下：

（1）创建脚本文件 scrapyd-deploy.bat，通过 bat 脚本运行 scrapyd-deploy 文件，实现项目打包处理。

（2）修改 Scrapy 项目文件 scrapy.cfg 的配置内容，根据部署方式配置属性 url 和 project。

（3）新建 CMD 窗口，将窗口路径切换到某个文件夹，输入 Scrapyd 服务开启指令 scrapyd。

（4）再次新建 CMD 窗口，将窗口路径切换到项目所在的文件夹，根据部署方式输入相应的打包指令，把项目打包到 Scrapyd 服务里。

（5）使用 Requests 模块调用 Scrapyd 的 API 接口，在 Scrapyd 服务里实现 Scrapy 项目的调度和管理。

26.2.2　Gerapy 爬虫管理框架

Gerapy 是一款分布式爬虫管理框架，它在 Scrapyd 和 Scrapyd-Client 的基础上进行封装，让 Scrapy 项目管理实现可视化操作。简单来说，Gerapy 是一个使用 Django 开发的管理系统，它可以部署和管理 Scrapy 项目。

使用 Gerapy 之前，需要在 Python 的开发环境下安装 Gerapy 框架，安装方式可以使用 pip 指令完成，指令如下：

```
# pip 在线安装 Gerapy
pip install Gerapy
```

Gerapy 安装成功后，在 D 盘创建文件夹 MyGerapy，打开 CMD 窗口并将当前路径切换到文件夹 MyGerapy 所在路径，然后输入指令 gerapy，CDM 窗口就会显示 Gerapy 框架的管理指令，如图 26-25 所示。

```
D:\MyGerapy>gerapy
Usage:
  gerapy init [--folder=<folder>]
  gerapy migrate
  gerapy createsuperuser
  gerapy runserver [<host:port>]
```

图 26-25　管理指令

Gerapy 框架共有 4 个指令，它们只适用于 Gerapy 框架管理，说明如下：

- gerapy init 是初始化 Gerapy 系统，创建一个新的 Gerapy 系统，用于部署和管理 Scrapy 项目。
- gerapy migrate 是对 Gerapy 系统的数据库进行初始化，在数据库里建立相关的数据表。
- gerapy createsuperuser 是在 Gerapy 系统创建超级管理员。
- gerapy runserver 是运行 Gerapy 系统。

在当前的 CMD 窗口下创建新的 Gerapy 系统，由于 CMD 窗口的路径是指向文件夹 MyGerapy，因此新建的 Gerapy 系统将会在文件夹 MyGerapy 里生成，如图 26-26 所示。



图 26-26　新建 Gerapy 系统

Gerapy 系统创建后，下一步是对系统的数据库进行初始化。将 CMD 窗口的路径切换到文件夹 gerapy 并输入 gerapy migrate，在 CMD 窗口可以看到数据表的创建信息，并且在文件夹 gerapy 里生成数据库文件 db.sqlite3，如图 26-27 所示。

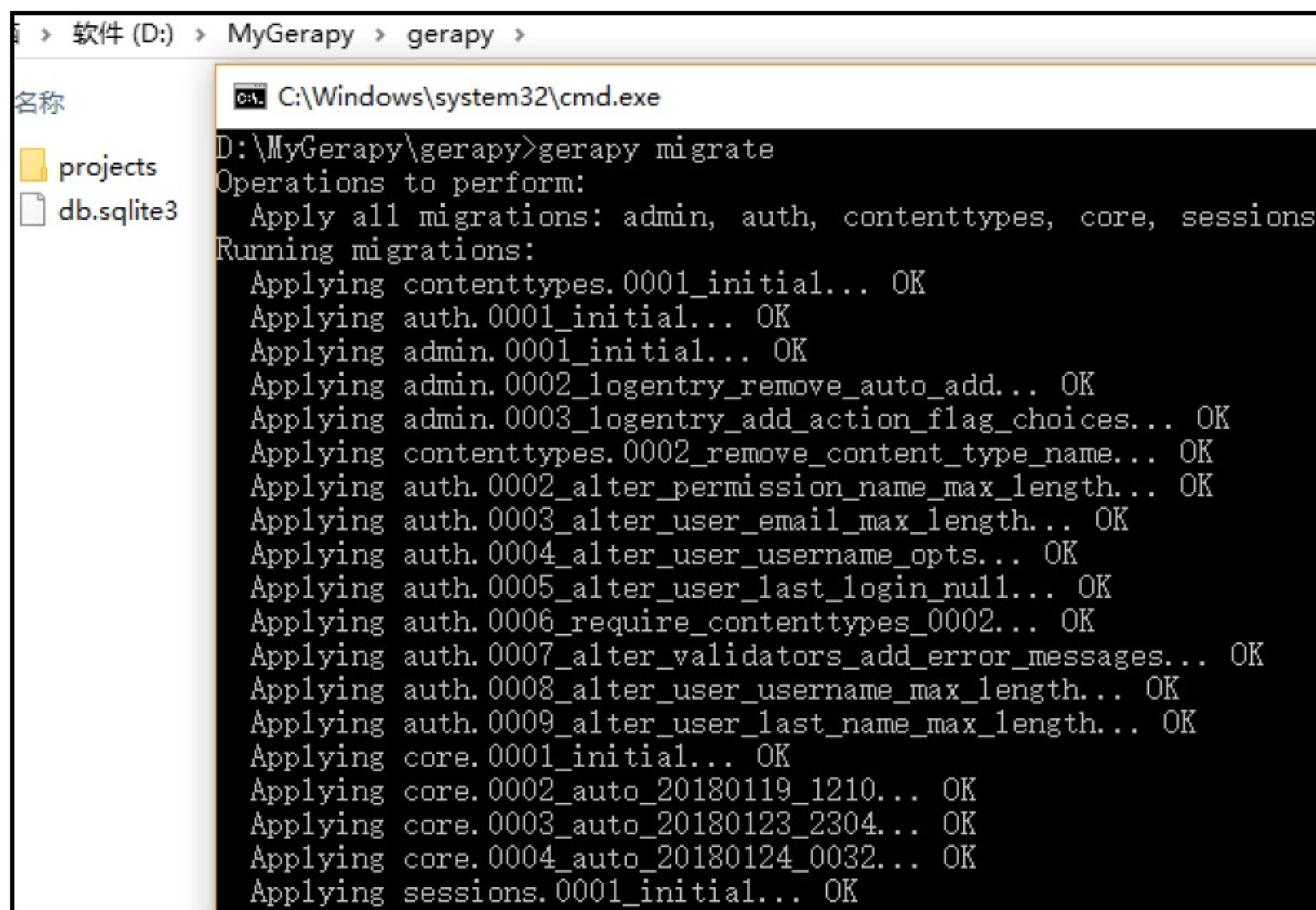


图 26-27　初始化数据库

完成上述操作后，就可以运行 Gerapy 系统，在 CMD 窗口下输入 gerapy runserver，并保证 CMD 窗口的路径是指向文件夹 gerapy，如图 26-28 所示。

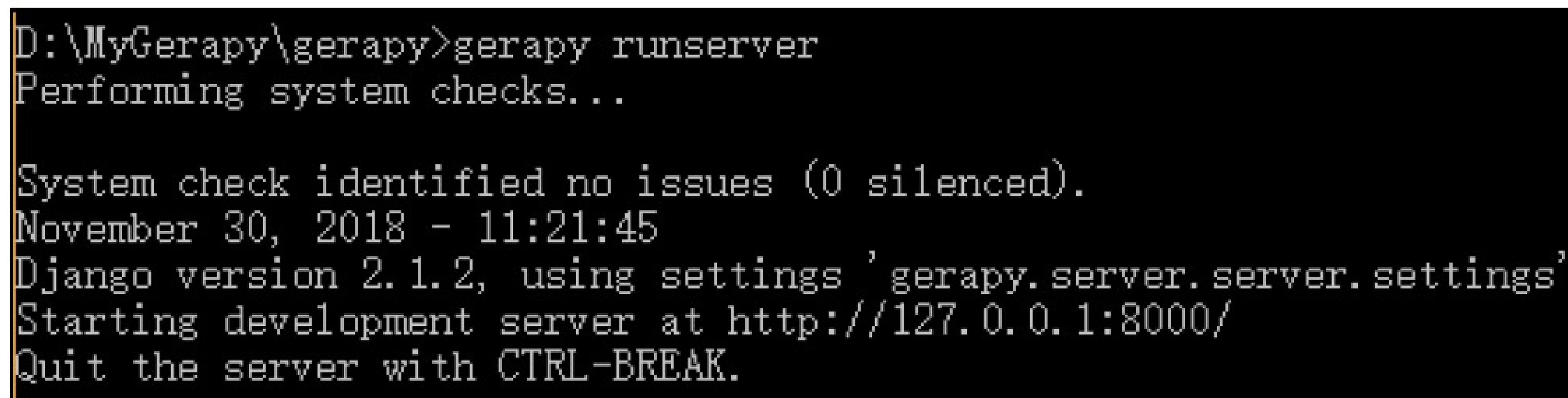


图 26-28　运行 Gerapy 系统

系统运行后，在浏览器上访问 http://127.0.0.1:8000/即可打开系统首页，将系统语言切换成中文模式，如图 26-29 所示。



图 26-29　系统首页

首页的右侧是显示当前系统所记录的主机和项目信息，这是一个简单的统计功能；而在首页的左侧分为主机管理和项目管理，这是管理系统的主机和项目。

主机管理是在系统里添加 Scrapyd 服务，比如现有多台主机并且每台主机已安装 Scrapyd 服务，为了统一管理这些主机，可以在 Gerapy 系统里添加主机信息，实现所有 Scrapyd 服务的调度和管理。在主机管理页面里，单击创建按钮即可添加主机信息，如图 26-30 所示。

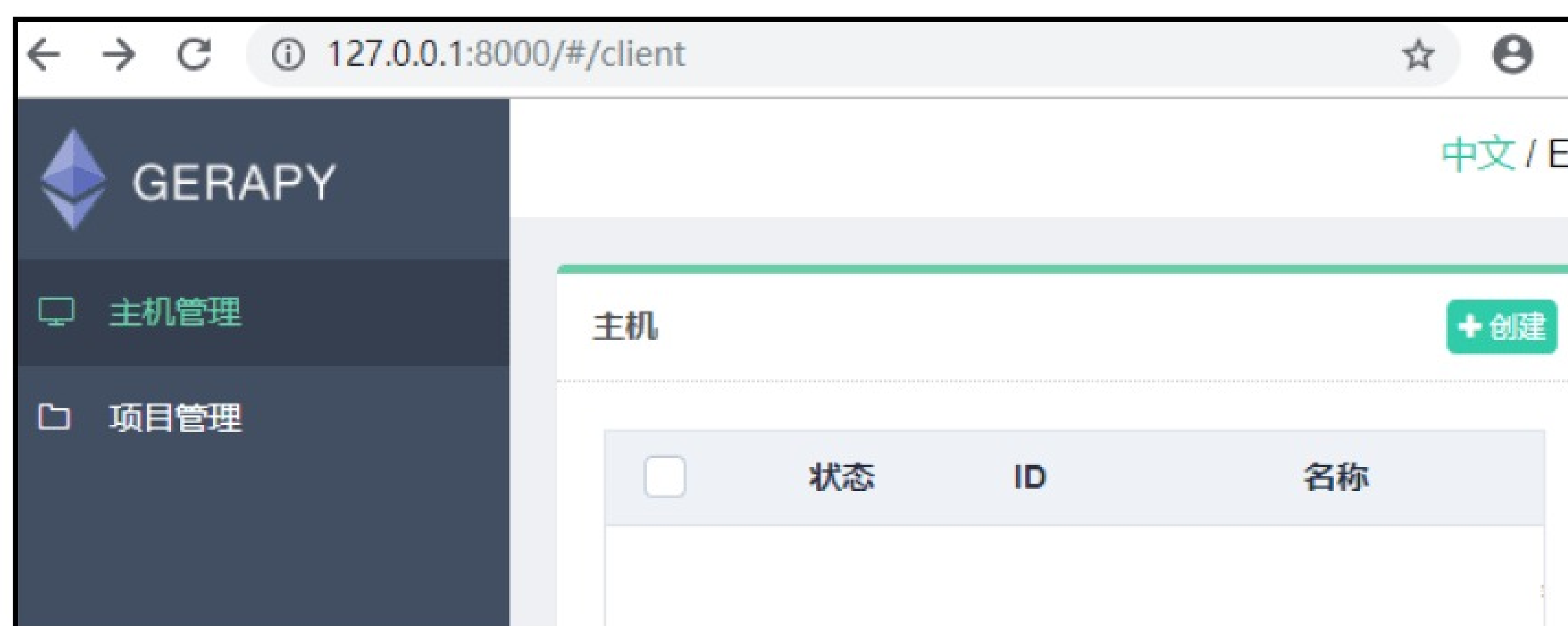


图 26-30　添加主机信息

在主机创建页面里，分别输入主机的名称、IP 和端口即可。主机名称可以自行命名；IP 是指 Scrapyd 服务所在主机的 IP 地址；端口是指 Scrapyd 服务所占用的端口，一般默认值为 6800。以本地的 Scrapyd 服务为例，创建主机信息如图 26-31 所示。

图 26-31　创建主机

单击“创建”按钮后，Gerapy 系统会将相关信息保存到数据库文件 db.sqlite3 中。返回到主机管理页面，可以看到刚创建的主机显示在主机管理页面，主机状态显示为错误，这说明当前主机尚未开启 Scrapyd 服务。当我们在本地开启 Scrapyd 服务，并且刷新主机管理页面后，主机状态显示为正常，如图 26-32 所示。



图 26-32　主机状态

下一步在项目管理中创建项目信息，使 Scrapy 项目与主机管理的 Scrapyd 服务相互结合，实现 Scrapy 项目部署。在 Gerapy 系统所在文件夹 gerapy 的 projects 文件夹里放置项目 douban，项目 douban 无需做打包处理，因为 Gerapy 系统已提供项目打包功能。放置项目 douban 后，再次刷新项目管理页面，可以看到项目 douban 信息，如图 26-33 所示。

名称	描述	打包	可配置	打包时间	操作
douban		X	X		编辑 部署 删除

图 26-33　项目 douban 信息

从项目管理页面看到，项目 douban 的打包和可配置都处于一个尚未激活状态。我们单击“部署”按钮，进入项目部署页面，输入描述内容并单击“打包”按钮即可完成项目打包处理，如图 26-34 所示。

项目打包成功后，单击图 26-34 的“部署”按钮即可将项目 douban 部署到名为 MyScrapyd 的 Scrapyd 服务。如果主机管理多个 Scrapyd 服务，在项目部署页面也会显示相应的 Scrapyd 服务，这样可将一个项目部署到多台主机，如图 26-35 所示。



图 26-34　项目打包处理

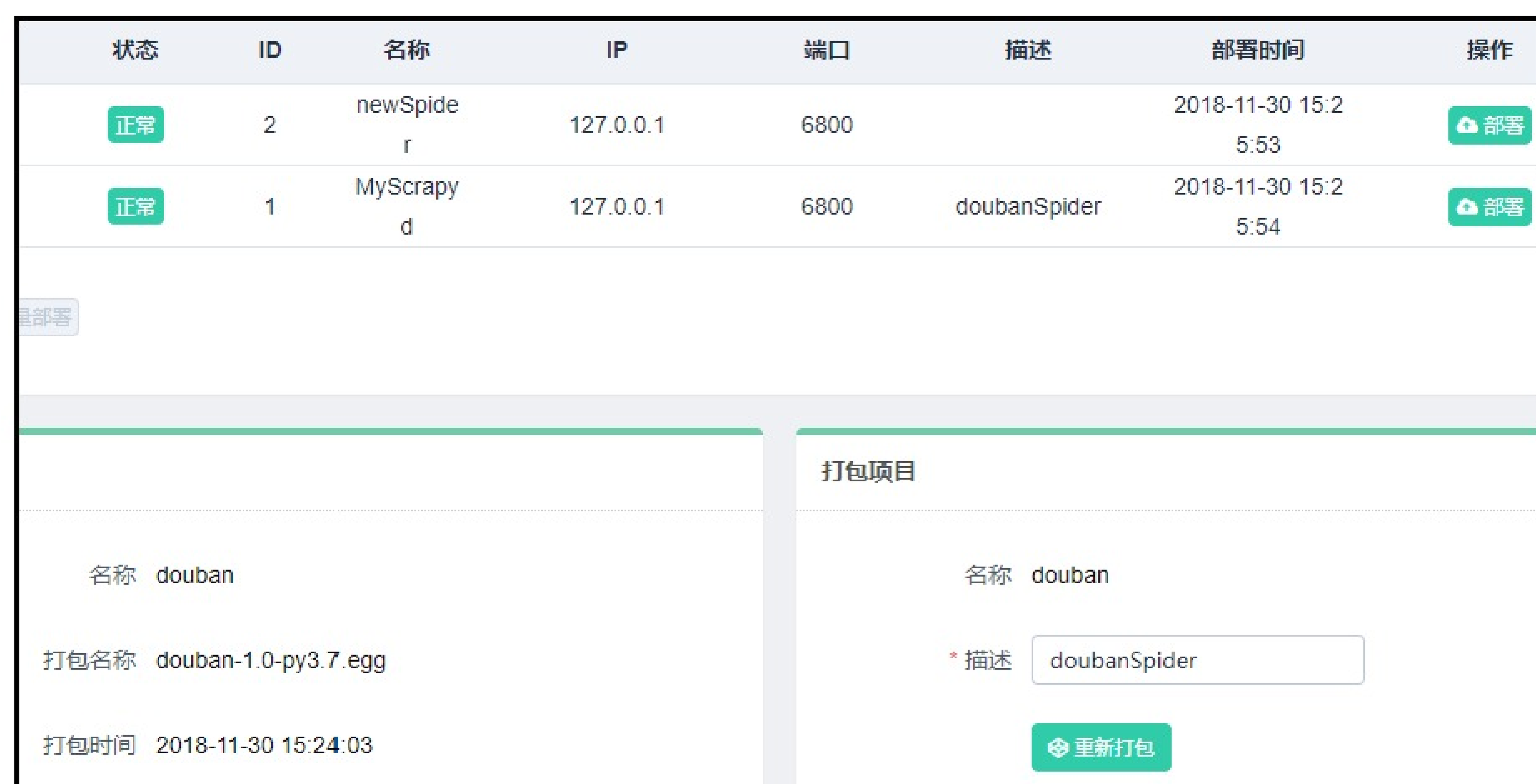


图 26-35　项目部署

如果项目 douban 的功能发生修改，需要在项目部署页面将项目重新打包处理，并且还要重新部署到相应的 Scrapyd 服务。

项目打包和部署已执行完毕，最后在 Gerapy 系统上运行项目。单击主机管理，进入主机管理页面并单击名为 MyScrapyd 的调度按钮，如图 26-36 所示。

状态	ID	名称	IP	端口	认证	操作
正常	2	newSpider	127.0.0.1	6800	X	编辑 调度 删除
正常	1	MyScrapyd	127.0.0.1	6800	X	编辑 调度 删除

图 26-36　主机管理页面

在 Scrapyd 服务页面里，隶属于当前 Scrapyd 服务的所有 Scrapy 项目都以列表的形式呈现。当运行项目 douban 的 Spider 程序时，页面就会生成程序的运行状态和运行信息，如图 26-37 所示。

ID	名称	操作
1	Movie	运行

爬虫名称: Movie　任务: e1a9de14f47611e8b894002324f7b942　取消　等待

爬虫名称: Movie　任务: 6eb3918cf47611e8b79a002324f7b942　开始时间: 2018-11-30 16:03　结束时间: 2018-11-30 16:03　完成

'movieId': '20438964'}]}

图 26-37　运行 Spider 程序

此外，Gerapy 系统还可以对主机和项目进行编辑和删除操作，详细的操作过程就不再一一讲述，读者可根据实际需求自行配置。

26.3 本章小结

爬虫的上线部署是为了让使用者方便使用爬虫程序，本章分别讲述了非框架式爬虫和框架式爬虫的部署方案。

非框架式爬虫是指爬虫程序使用非框架式开发，也就是使用 urllib、requests 和 BeautifulSoup 等这类爬虫模块实现的爬虫程序，整个过程不涉及爬虫框架，如 Scrapy 和 PySpider 等。非框架式爬虫的部署方式如下所示：

（1）创建可执行程序，使用者只需双击程序即可运行爬虫。

（2）制定任务计划程序，利用计算机的任务管理器来控制爬虫程序的运行时间，无须使用者操作，实现自动化爬虫。

（3）创建服务程序，利用计算机的服务程序来运行爬虫程序，只要计算机没有关机，爬虫每时每刻都在运行。

框架式爬虫是指使用 Scrapy 或 PySpider 等爬虫框架开发的爬虫程序，这种爬虫程序有自身的运行方式。对于这类爬虫只能通过特定的工具部署，以 Scrapy 框架为例，部署工具有 Scrapyd 和 Gerapy，两者说明如下：

（1）Scrapyd 是一款用于管理 Scrapy 爬虫的部署和运行的服务，由 Scrapy 的开发者开发，它是通过 API 接口来部署或者控制 Scrapy 项目，可以管理多个项目，并且每个项目还可以上传多个版本，但只有最新的版本会被使用。

（2）Gerapy 是一款分布式爬虫管理框架，它在 Scrapyd 和 Scrapyd-Client 的基础上进行封装，让 Scrapy 项目管理实现可视化操作。简单来说，Gerapy 是一个使用 Django 开发的管理系统，它可以部署和管理 Scrapy 项目。

第 27 章

反爬虫的解决方案

27.1 常见的反爬虫技术

反爬虫一直都是爬虫开发最让人头疼的问题。相信很多开发者会遇到这样的情况，在开发过程中，爬虫程序成功通过测试，但项目上线就会出现各种问题，比如 HTTP 请求出现异常或者无法从响应内容爬取目标数据等，这种“程序开发正常，上线出异常”的情况是因为网站设置了反爬虫机制。

不同类型的网站反爬机制也不同，判断一个网站是否有反爬机制需要根据网站设计架构、数据传输方式和请求方式等多个方面来评估。下面列出常用的反爬虫技术。

（1）用户请求的 Headers。
（2）用户操作网站行为。
（3）网站目录数据加载方式。
（4）数据加密。
（5）验证码识别。

网站设置反爬机制不代表不能爬取数据，正如“你有张良计，我有过墙梯”，每种反爬虫机制都有对应的解决方案。

1. 基于用户请求的 Headers

从用户请求的 Headers 反爬虫是最常见的反爬虫策略。很多网站会对 Headers 的 User-Agent 进行检测，还有一部分网站会对 Referer 进行检测（一些资源网站的防盗链就是检测 Referer）。如果遇到了这类反爬虫机制，可以在爬虫代码中添加 Headers 请求头，将浏览器的请求信息以字典的数据格式写入爬虫的请求头；对于检测 Headers 的反爬虫，在爬虫发送请求中修改或者添加 Headers 就能很好地解决。

2. 基于用户操作网站的行为

还有一部分网站是通过检测用户行为来判断用户行为是否合规，例如同一 IP 短时间内多次访问同一页面或者同一账户短时间内多次进行相同操作。网站服务器会根据已设定的判断条件判断访问间隔是否合理，从而达到检测用户行为是否合理。

遇到用户行为判断这种情况，可使用 IP 代理，IP 可以在 IP 代理平台上通过 API 接口获取，每请求几次更换一个 IP，这样就能很容易地绕过第一种反爬虫。

还有一种解决方案是在每次请求间隔几秒后再发送下一次请求。只需在代码里面加一个延时功能就可以简单实现，有些有逻辑漏洞的网站可以通过请求几次、退出登录、重新登录、继续请求来绕过同一账号短时间内不能多次进行相同请求的限制。

3. 基于网站目录数据加载

上述几种情况大多都出现在静态页面，还有一部分网站是由 Ajax 通过访问接口的方式生成数据加载到网页。遇到这样的情况，首先分析网站设计，找到 Ajax 请求，分析具体的请求参数和响应的数据结构及其含义，在爬虫中模拟 Ajax 请求，就能获取所需数据。

4. 基于数据加密

在很多情况下没有想象中那么完美，能够直接模拟 Ajax 请求获取数据固然是极好，但部分网站会把请求的参数加密处理。这种情况可先找到加密代码，加密代码主要是使用 JavaScript 实现的，分析代码的加密方式，然后在爬虫代码中模拟其加密处理，再发送请求。这是最优解决方案，但花费时间较多，难度相对较大。

另一种解决方案是使用 Selenium+PhantomJS 框架（自动化测试技术），调用浏览器内核，利用 Selenium 模拟人为操作网页并触发页面中的 JS 脚本，完整地实现自动化操作网页，数据是从网页上自动获取的。这套框架几乎能绕过大多数反爬虫，因为 PhantomJS 是一个没有界面的浏览器。Selenium+PhantomJS 能解决很多事情，而且用途很大。但是在爬虫中，不到万不得已的地步，一般不支持使用这种方式，因为这种方式已经是自动化测试的范畴，有点偏离了爬虫开发。

5. 基于验证码识别

最有效的反爬虫技术就是验证码，目前对复杂的验证码还没有做到很好地识别验证，只能通过第三方平台处理或者 OCR 技术识别。当然，不是说有验证码就不能做爬虫，只是目前在验证码的问题上还没有一个很完美的解决方案。

27.2 基于验证码的反爬虫

验证码是反爬虫机制里面最常用的手段之一，在本书第 12 章已对验证码识别做了详细的讲述，在本节中，我们来总结爬虫中出现验证码的情况以及处理方式。

27.2.1　验证码出现的情况

1．登录页面设有验证码识别

登录页面是验证码出现频率最高的页面，它不仅能提高用户的安全性，而且还起到反爬虫作用。以 B 站为例，单击用户登录就会出现带滑动验证码的登录页面，如图 27-1 所示。



图 27-1　带验证码的登录页面

2．特殊请求设置验证码

在网页里对某些关键操作设置验证码，这种验证码出现方式虽然很影响用户体验，但可以很好地防止数据泄漏。主要应用在政府网站或商品抢购活动等，如查询企业信用信息（www.gsxt.gov.cn/index.html），输入企业名称并单击查询按钮就会出现验证码，如图 27-2 所示。



图 27-2　特殊请求设置验证码

3．网络环境导致验证码出现

网络环境的不稳定因素主要是本地网络的 IP 地址已被网站服务器列入异常名单。列入异常名单的方式有很多，比如访问过于频繁，在 20.4 节中已有介绍，如果微博爬取的间隔时间太短就会出现验证码；再比如代理 IP 的安全性也会导致验证码出现，网络上大多数的免费代理 IP 都是不安

全的，若使用这些免费代理 IP，很容易遭到反爬虫拦截。

27.2.2 解决方案

对于上述验证码，这里总结归纳几个可行的解决方案，说明如下：

1．使用 OCR 技术识别

目前 OCR 技术已逐步趋向成熟，并且扩展性强，能满足开发人员的二次开发，但开发难度较大，若没有经过二次开发或验证码训练，仅靠 OCR 自身的识别能力，识别准确率会非常低。

2．使用第三方平台 API 接口识别

目前第三方平台已经可以识别各种类型的验证码，而且准确率高和响应速度快，但每次使用需要收取相应的费用。

3．人为识别验证码

爬虫程序在运行过程中，若出现验证码，会首先下载验证码图片，并且使用 input 函数使程序处于等待状态，然后人为识别验证码，把验证码内容通过 input 函数传递到程序里，从而解决验证码识别问题。使用此方法需要掌握各种验证码的生成原理，第 19 章的 12306 网站用户登录就是使用的此方法解决验证码识别问题。

4．使用 Selenium 控制程序实现识别验证码

如果页面出现验证码，使用 Selenium 模拟打开浏览器访问页面，并且爬虫程序必须要处于等待状态，然后人为在浏览器上完成验证码识别并退出爬虫程序的等待状态，让程序获取浏览器的 Cookies 信息并往下执行。由于浏览器已完成验证码识别，因此在后续的 HTTP 请求中加入 Cookies 即可解决验证码问题。这种方法的代码格式相对固定，如下所示：

```
from selenium import webdriver
# 需要安装 pywin32 模块
import win32api, win32con
driver = webdriver.Chrome()
driver.get("https://www.baidu.com/")
# 生成 Windows 提示框，让程序处于等待状态
# 当点击提示框的确定按钮，程序即可往下执行
win32api.MessageBox(0, "请识别验证码","提示",win32con.MB_OK)
# 获取浏览器的 Cookies
cookie = driver.get_cookies()
driver.quit()
# Cookies 格式化
cookie_dict = {}
for i in cookie:
   cookie_dict[i['name']] = i['value']
```

5．从浏览器复制 Cookies 并写入程序

在某些网站，用户登录一次后就会永久保存用户登录状态，只要不退出用户登录或不删除浏览器的历史记录，无论关闭浏览器还是重启计算机，当再次访问网站的时候，网站都会显示上次登

录的用户信息。对于这类网站，它的 Cookies 永久有效，可以从开发者工具里复制 Cookies 信息并写入程序，程序每次发送请求只需设置 Cookies 即可跳过用户登录，从而避开用户登录出现验证码识别的问题。这种方法只需将 Cookies 转化成字典格式即可使用，如下所示：

```
# 从浏览器的开发者工具获取 Cookies
cookie = 'bid=GYJXyB0Wc; user id=dc6354c'
# Cookies 格式化
cookie_dict = {}
# 将 Cookies 转换字典格式
for i in cookie.split(';'):
    key = i.split('=')[0]
    value = i.split('=')[1]
    cookie_dict[key] = value
print(cookie_dict)
```

27.3　基于请求参数的反爬虫

爬虫是通过 HTTP 请求来获取目标数据的，而请求参数是 HTTP 请求必不可少的构成部分，因此很多反爬虫机制都设置了请求参数。

27.3.1　请求参数的数据来源

结合本书的实战项目，这里我们总结出请求参数的数据来源方式。

1．参数值为固定可选值

这类请求参数是很容易辨认的，同一个请求，其请求参数可能是固定不变或者按一定的规律变化。比如猫眼电影的 Top100 排行榜（http://maoyan.com/board/4?offset=0），请求参数 offset 以 0、10 和 20 的规律递增，分别代表第 1 页、第 2 页和第 3 页。

2．参数值来自其他请求的响应内容

如果参数值来自其他请求的响应内容，需要将参数值在各个请求的响应内容里查找。比如请求 A 的请求参数是从请求 B 的响应内容里获取，而请求 B 的请求参数是从请求 C 的响应内容获取，这三个请求就形成一种递进关系。若要在爬虫里获取请求 A 的响应内容，首先获取请求 C 的响应内容，得出请求 B 的请求参数，然后对请求 B 发送 HTTP 请求，从响应内容获取请求 A 的请求参数，最后才能获取请求 A 的响应内容。这种方式是爬虫开发中最为常见的，如 12306 用户登录、QQ 音乐的歌曲下载等。

3．参数值经过 JS 处理

JS 处理方式有加密、混淆或数据转换等，这是一种有效的反爬虫机制，同时能提高用户账号的安全性。第 20 章的微博用户登录功能就涉及到 JS 加密，把用户账号和密码都进行加密处理，并且加密的 JS 代码经过压缩处理，还有一些网站会对 JS 代码进行混淆处理，让爬虫开发者难以分析 JS 处理过程，这一系列的防御机制为爬虫开发带来一定的难度，从而起到反爬虫的作用。

4．参数值为特殊值

判断参数是否为特殊值，需要判断参数值的数据结构。比如参数值由一串数字组成并且以 15 开头，这串数字可能是一个时间戳，以时间戳作为请求参数，代表该请求具有时效性，网站会根据时间戳的时间和实际时间进行对比，若时间戳的时间符合要求，说明当前请求合法，反之则判为异常请求。

如果参数值是一串不规则的数据并且无法在其他请求的响应内容里获取，很可能它是来自 Cookies 信息，第 18.2 节的 QQ 歌曲下载就是使用 Cookies 信息作为请求参数的。

综上所述，请求参数的数据来源主要有：参数值为固定可选值、参数值来自其他请求的响应内容、参数值经过 JS 处理以及参数值为特殊值。

27.3.2 请求参数的查找

根据请求参数的类型，可以总结出请求参数的查找方法，分述如下。

1．查找请求参数的变化规律

查找请求参数的变化规律是分析请求参数最常用的手段，它能通过变化规律找出请求参数的含义与作用。查找变化规律需要分析同一个请求参数的不同参数值，如猫眼电影 Top100 排行榜的请求参数 offset，当单击网页下方的页数时，请求参数 offset 随之变化，其规律是以 0、10、20……依次递增，分别代表第 1 页、第 2 页、第 3 页……如图 27-3 所示。



图 27-3　请求参数变化规律

除了分析请求参数的参数值变化之外，还可以对请求参数的命名进行分析。比如 offset 的中文翻译为偏离的，其中文意思代表排行榜电影的偏离量，若将 100 部电影以 10 为单位进行划分，共有 100/10=10 个单位，单位总数分别与网页的页数相互对应，每个单位数为 10 代表了每页有 10 部电影。

2．从其他请求信息查找

从其他请求信息查找请求参数也是分析请求参数的常用手段，查找方法只需将参数值在所有请求的响应内容里依次查找即可。但有时候会出现特殊情况，请求参数是来自前面的请求信息，如

12306 的预订车票请求，该请求参数 train_date 是来自车次信息请求，而预订车票请求和车次信息请求是在不同的页面，即两者不是同一页面生成的请求信息，这种跨页面的数据传递也能起到反爬虫的作用。

对于请求参数的跨页面传递问题，如果当前所有请求都无法找到请求参数，可以尝试从上一页面的请求信息里查找，查找过程中需要对每个数据的命名和结构进行分析和对比，总的来说，请求参数的查找需要耐性和细心分析。

3．实现 JS 代码处理过程

请求参数经过 JS 处理是反爬虫里最有效的手段之一，因为破解并还原 JS 处理过程是相当耗时的，从而延长爬虫开发周期。破解并还原 JS 处理有两种方法，说明如下：

直接解读 JS 代码，在爬虫里还原 JS 处理并生成请求参数，这种方法非常考验爬虫工程师的 JS 熟练程度，第 20 章的微博登录就是直接解读并还原 JS 处理。

利用 Python 的 JS 运行库模拟运行 JS 代码，这种方法只需在掌握 JS 代码的输入和输出即可，无需解读具体的处理过程。Python 模拟运行 JS 的库有 PyExecJS 和 pyv8，以 PyExecJS 为例，在 CMD 窗口输入安装指令：pip install PyExecJS。JS 代码是以字符串的形式表示，然后调用 PyExecJS 库的方法即可运行 JS 代码，如下所示：

```
import execjs
def exec_js(x, y):
    # 编译 JS 代码
    ctx = execjs.compile("""
       function add(x, y) {
           return x + y;
           }""")
    # 执行代码
    return ctx.call("add", x, y)
print(exec_js(1, 2))
```

上述代码简单调用了 JS 的 add 函数，函数参数 x、y 是在调用的过程中以变量的形式传入。在爬虫开发过程中，当遇到 JS 处理的请求参数时，只需将相应的 JS 代码以字符串的形式写入程序，然后使用 PyExecJS 库即可得出处理后的请求参数。

4．利用浏览器动态加载

如果请求参数确实过于复杂而且难以实现，还可以使用 Requests-HTML、Selenium 或 Splash 等模块实现浏览器动态加载。只需将爬取的网址传递给浏览器即可爬取目标数据，如果涉及到表单提交，可以使用 Selenium 或 Splash 模拟操作浏览器，输入表单数据并单击提交按钮即可实现。

虽然这种方法可以轻松实现数据爬取，但在浏览器加载网页的时候，它会自动加载网页的全部请求信息，因此访问速度比单个 HTTP 请求的访问速度要慢。

27.4　基于请求头的反爬虫

在请求头里设置反爬虫机制也是常见的手段之一，第 2.2 节已对请求头做了详细的介绍，本节

根据实战项目来总结一下请求头的反爬虫机制。

1. 检测请求头的固定属性

固定属性是指请求头的属性具有固定值或者按一定的规律变化，如 User-Agent、Connection、Host 或 Referer 等基本属性，这些属性在每个请求信息里都能轻易找到，并且属性值相对固定。

在实战项目 23.3 节里，爬取豆瓣电影评论必须添加请求头的 User-Agent 属性，否则在爬取过程中就会提示网页 403 错误信息，这说明网站服务器对请求头属性 User-Agent 设置检测功能。

如果爬虫里加入异步并发功能，在同一时间内发送多条 HTTP 请求并且出现了异常信息，且提示无法建立新的请求连接（failed to establish a new connection），这是请求队列已塞满 HTTP 请求的原因，对此需在每个请求里设置请求头属性 Connection，属性值为 close，这样可将已完成的 HTTP 请求关闭释放。

请求头属性 Host 是根据请求地址 URL 的变化而变化。一般情况下，Host 的属性值与 URL 的域名信息一致，假如请求头的 Host 属性值与 URL 的域名不相符，那么该请求可能无法获取正确的响应内容。这种反爬虫机制常用于分类性的网站，如美团或链家等，此网站是以城市进行分类，不同的城市有不同的域名，如图 27-4 所示。



图 27-4　域名信息

属性 Referer 说明当前请求来自于哪个页面的 URL，反爬虫机制引入属性 Referer 是利用 HTTP 防盗链技术实现。在 HTTP 协议中，当一个网页跳到另一个网页时，HTTP 的请求头都会带有属性 Referer，网站服务器通过检测属性 Referer 是否在服务器的域名名单，如果 Referer 不在域名名单中，当前请求会被判为非法请求，无法获取正确的响应内容。

此外，请求头还有很多特殊且固定的属性，如 Upgrade-Insecure-Requests 或 X-Requested-With 等，其特殊性在于这些属性并不经常出现，当请求头出现这些属性的时候，其属性值大多数是一个固定值，如图 27-5 所示。

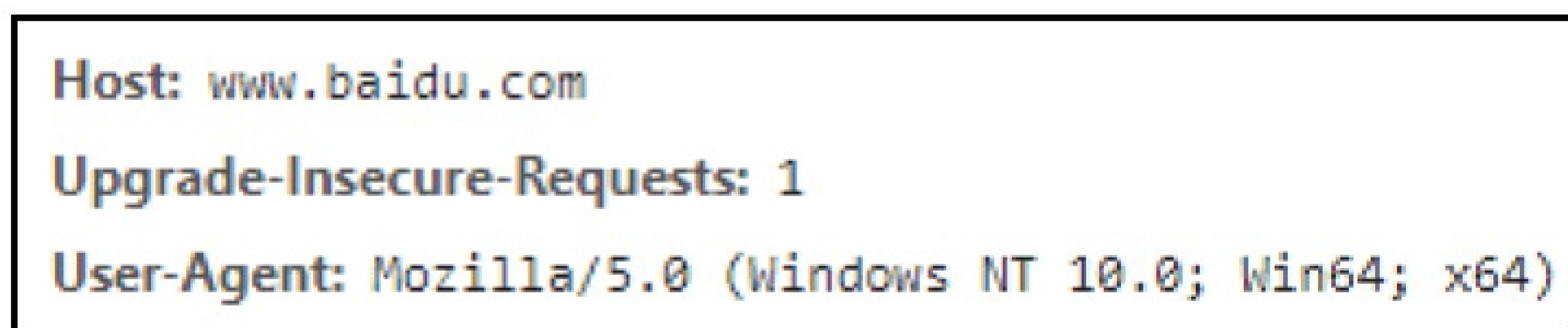


图 27-5　请求头信息

2. 检测请求头的可变属性

可变属性是指请求头的属性以一定的计算方式变换属性值，这类属性一般都是自行命名，其

作用是生成反爬虫机制，而且反爬虫效果非常好。以一个例子加以说明，其请求头如图 27-6 所示。

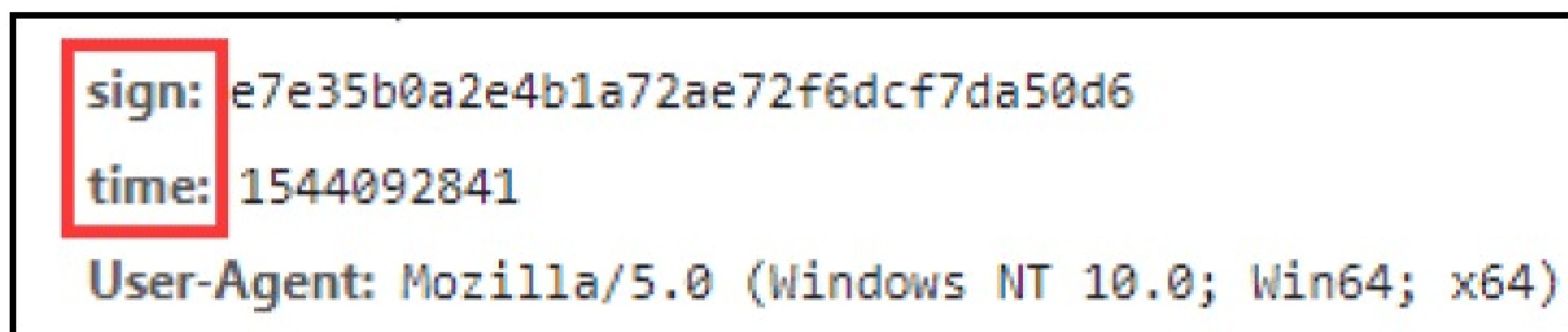


图 27-6　自定义请求头属性

从图 27-6 上看出，请求头属性 sign 和 time 是自定义属性，属性 time 是以时间戳的形式表示，代表当前请求的时间，这说明当前请求具有时效性。属性 sign 是一串不规则的字符串，说明字符串经过了加密处理，若要破解 JS 的加密过程，只能查找 JS 代码所在的请求信息，如图 27-7 所示。

```
function(key, index) {\n
    if (index == __WEBPACK_IMPORTED_MODULE_0_babel_runtime_core_js_object_keys___default()(subData).length -
1) {\n signs += key + \" is \" + subData[key] + \" & cm91dGVyYmV1dG8\";\n            } else {\n
signs += key + \" is \" + subData[key] + \" and \";\n            }\n         });\n         signs =
signs.split(\"\").reverse().join(\"\");\n         var headsData = {\n            time: users.time,\n
sign: this.$md5(signs).toLowerCase()\n         };\n         this.$refs[\"logindata\"].validate(function (valid)
{\n            if (valid) {\n               WEBPACK_IMPORTED_MODULE_2_axios___default.a.post(\"/user/login\",
formData, {\n                  headers: headsData\n               }).then(function (res) {\n                  if
(res.data.error != 0)
```

图 27-7　破解 JS 代码

分析图 27-7 上的代码得知，JS 代码里的请求头 headsData 定义了两个属性 time 和 sign，属性 sign 是将函数参数 key 进行反转、md5 加密和大小写转换等处理，若要得知参数 key 的数据内容，还要深入分析 JS 代码。

综上，请求头的属性根据属性值内容分为两类：固定属性和可变属性。在爬虫开发里，建议将请求头的固定属性都写入到请求头里；如果请求头含有可变属性，必须找出属性值的变化规律，然后由爬虫程序计算得出正确的属性值，最后将可变属性写入请求头并完成 HTTP 请求。

27.5　基于 Cookies 的反爬虫

Cookies 是网站为了辨别用户身份、进行 Session 跟踪而储存在用户本地终端上的数据，一个 Cookies 就是存储在用户主机浏览器中的文本文件。网站利用 Cookies 设置反爬虫机制的主要方式有：限制 Cookies 的访问频率（即限制用户的访问频率）和限制 Cookies 的时效性。

1. 限制 Cookies 的访问频率

在一定的时间内，网站设置了同一用户的访问上限，如果超出访问上限，网站会认为当前用户是爬虫程序或机器人，从而引发反爬虫机制。不同网站的访问上限各不相同，并且难以测出具体的上限数，爬虫开发者能通过延迟每个 HTTP 请求的发送间隔，比如每个请求之间的间隔设为 3 秒、5 秒以及 10 秒等，通过降低访问频率来避开反爬虫机制的检测。

2. 限制 Cookies 的时效性

限制 Cookies 的时效性主要通过 JS 处理，每次发送 HTTP 请求都会根据上一次请求的 Cookies 进行动态修改，改变后的 Cookies 将作为当前请求的 Cookies 信息，网站收到 HTTP 请求后，根据 Cookies 的处理逻辑去判断当前请求的 Cookies 是否正确。如果判断结果为正确，网站则返回正确的响应内容，否则触发反爬虫机制，提示 404 或 500 等异常信息。

以某国外网站（https://www.similarweb.com/category）为例，该网站在国内暂时无法访问，读者需自行处理。访问网站并打开开发者工具的 Network 选项卡，在 Doc 标签下找到网页的请求信息，并查看 Cookies 信息，如图 27-8 所示。

```
Cookie: sgID=43b3994a-f102-4d7e-9bfb-8421f1e885c9; .AspNetCore.Antiforgery.xd9Q-ZnrZJo=CfDJ8DmQhnTVHkJMqh
PArtVp3GiJsTkfX1PnELoNTabWDE1RL0ThIe90S5XuLa7jqAxMNwVpHpSp359J9dHuzo9rYqSah1wiwuWs21goWetr3Tvwa0D3LOlqVp
6GKwdM90imV-DF4CbO2Ual53_hoQg; _ga=GA1.2.228349126.1544148546; _gid=GA1.2.1110147088.1544148546; _gcl_au
1.1.1860064709.1544148546; user_num=nowset; _fbp=fb.1.1544148547264.1049076161; _vwo_uuid_v2=DE7772A0CB7
DA80314E42EC6E065140|a3c53ca9b7fa9c4b257377dbd040ab42; _vis_opt_s=1%7C; _vis_opt_test_cookie=1; _mkto_tr
id:891-VEY-973&token:_mch-similarweb.com-1544148551518-16299; D_IID=E85317E3-9239-3F74-8ADD-215547869C54
D_UID=6725FA60-BD98-37D9-A065-9952B48880FD; D_ZID=C0C2E046-0482-3057-A67C-8358761793A6; D_ZUID=D46F0FFD-
71-3006-A843-8EDF9CD3512B; D_HID=190993A6-2824-3463-AD7E-13FEB1A1D112; D_SID=45.114.164.111:kzhnXyY5/J/3
WiWYHR2OOzm/C5uwU5KrDgiNTyOCo; visitor_id597341=286517961; visitor_id597341-hash=f634c94aa135b47702fcbcc
03da180a5751435ce77e5343e049d350829a1356171637c13da38727b9391f3c0c26ccda955289e; loyal-user=%7B%22date%2
3A%222018-12-07T02%3A09%3A10.838Z%22%2C%22isLoyal%22%3Atrue%7D; intercom-id-e74067abd037cecbecb0662854f0
ee12139f95=d2de1193-c04d-4099-a955-30a5cb60b5b9; _pk_ses.1.fd33=*; _gat=1; sc_is_visitor_unique=rx861714
1544154789.95524C6C8BE44FB723A973E159753C19.2.2.1.1.1.1.1.1.1; _pk_id.1.fd33=c411e85798805d10.1544148634
2.1544154790.1544154296.
```

图 27-8 查看 Cookies 信息

当再次刷新网页并查看 Cookies 信息时，发现 Cookies 的 sc_is_visitor_unique 和 pk_id 属性与上一次请求的 Cookies 有所不同，两者的数据都加入了时间戳，使得每次请求的 Cookies 都会有所不同；并且 loyal-user 属性设有时间日期格式。

在爬虫开发中，如果是 Cookies 引发的反爬虫机制，可以采取 4 种应对方法：构建 Cookies 池、使用代理 IP、动态构建 Cookies 和利用浏览器获取 Cookies，说明如下：

1. 构建 Cookies 池

Cookies 池是将不同的 Cookies 以列表的形式表示，在每次发送 HTTP 请求之前，从 Cookies 池随机抽取某条 Cookies 信息作为 HTTP 请求的 Cookies，这样能降低同一条 Cookies 的访问频率，避免反爬虫机制的检测。

一般情况下，一台计算机在网站里只会有一个 Cookies 信息，若要使用同一台计算机生成多个不同的 Cookies，需要借助谷歌浏览器的无痕模式，以美团美食网站为例，具体操作如下。

打开新的谷歌浏览器，单击右上方的“自定义及控制”按钮，选中单击“打开新的无痕窗口”选项，如图 27-9 所示。

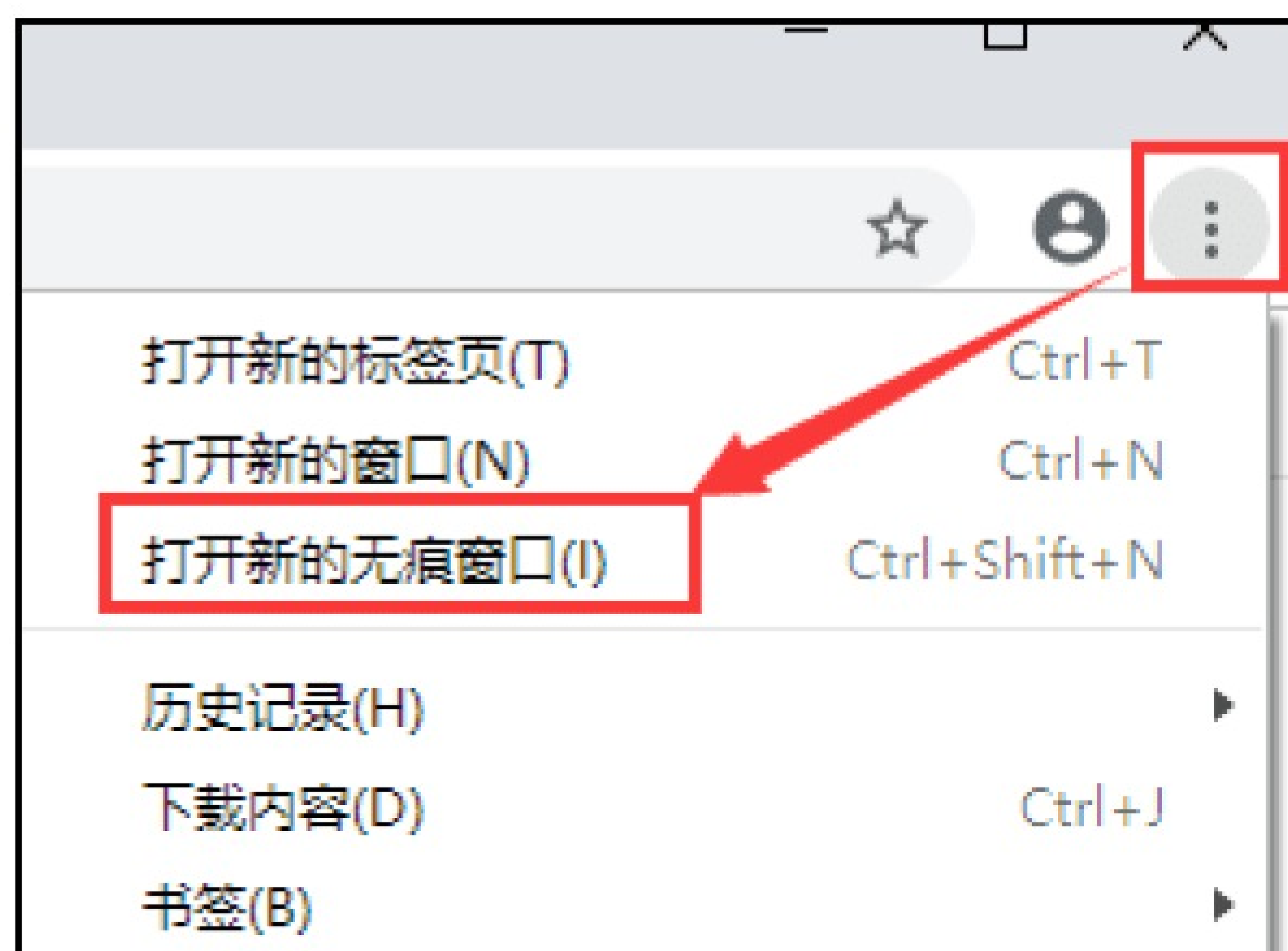


图 27-9　打开新的无痕窗口

在无痕模式下输入美团美食网站（https://gz.meituan.com/meishi/），并在开发者工具中获取 Cookies 信息，如图 27-10 所示。

```
Cookie: client-id=4f51a83f-11a9-47d9-81a3-5ff986a5d474; uuid=01fc565f-bcbd-4f43-831c-dafb4ae51465; _lxsdk_cuid=16787653321c8-07295e9515380e-8383268-1fa400-16787653321c8; _lxsdk=16787653321c8-07295e9515380e-8383268-1fa400-16787653321c8; _lxsdk_s=16787653323-39d-c08-1c2%7C%7C2
Host: gz.meituan.com
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36
```

图 27-10　第一次获取 Cookies

在第二次获取新的 Cookies 之前，需要将谷歌浏览器全部关闭，然后重新打开新的无痕窗口，再次访问美团美食网站获取新的 Cookies，如图 27-11 所示。

```
Cookie: client-id=3625a185-cb1d-4e58-a588-fcb833df0509; uuid=ab2b294e-4095-43fd-8aff-335be7a8572b; _lxsdk_cuid=167876fb2f92-0d267035bc82f2-8383268-1fa400-167876fb2fac8; _lxsdk=167876fb2f92-0d267035bc82f2-8383268-1fa400-167876fb2fac8; _lxsdk_s=167876fb2fa-2bf-8e2-4cc%7C%7C2
Host: gz.meituan.com
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36
```

图 27-11　第二次获取 Cookies

对比图 27-10 和图 27-11 发现，在无痕模式下获取的 Cookies 信息各不相同，这种方法构建 Cookies 池是可行的。如果 Cookies 信息记载了用户登录信息，那么构建 Cookies 池就需要不同的

用户账号。

2．使用代理 IP

代理 IP 是解决 Cookies 反爬虫机制的最有效手段，它可以解决多用户使用同一个 IP 地址的安全问题，使用代理 IP 可以真实模拟多用户在访问网站。虽然谷歌的无痕窗口可以得到多个不同的 Cookies 信息，但这些 Cookies 可能存储了计算机的 IP 地址等信息，如果网站服务器对这些信息进行检测，也很容易引发反爬虫机制。

网络上有专门提供代理 IP 服务的网站，在第 21.4 节已讲述了代理 IP 的使用，也是调用网站提供的 API 接口实现，详细的使用方法可查看相关网站提供的文档说明。

3．动态构建 Cookies

如果 Cookies 设置了时效性，我们可以解读 JS 代码来获取 Cookies 处理逻辑，在一个网页里，JS 代码主要存放在 Network 选项卡的 JS 标签和 Doc 标签中。以上述国外网站（https://www.similarweb.com/category）为例，在 JS 标签和 Doc 标签中都能找到 Cookies 的处理代码，由于 Cookies 的处理代码较多，此处只列举部分代码，如图 27-12 所示。

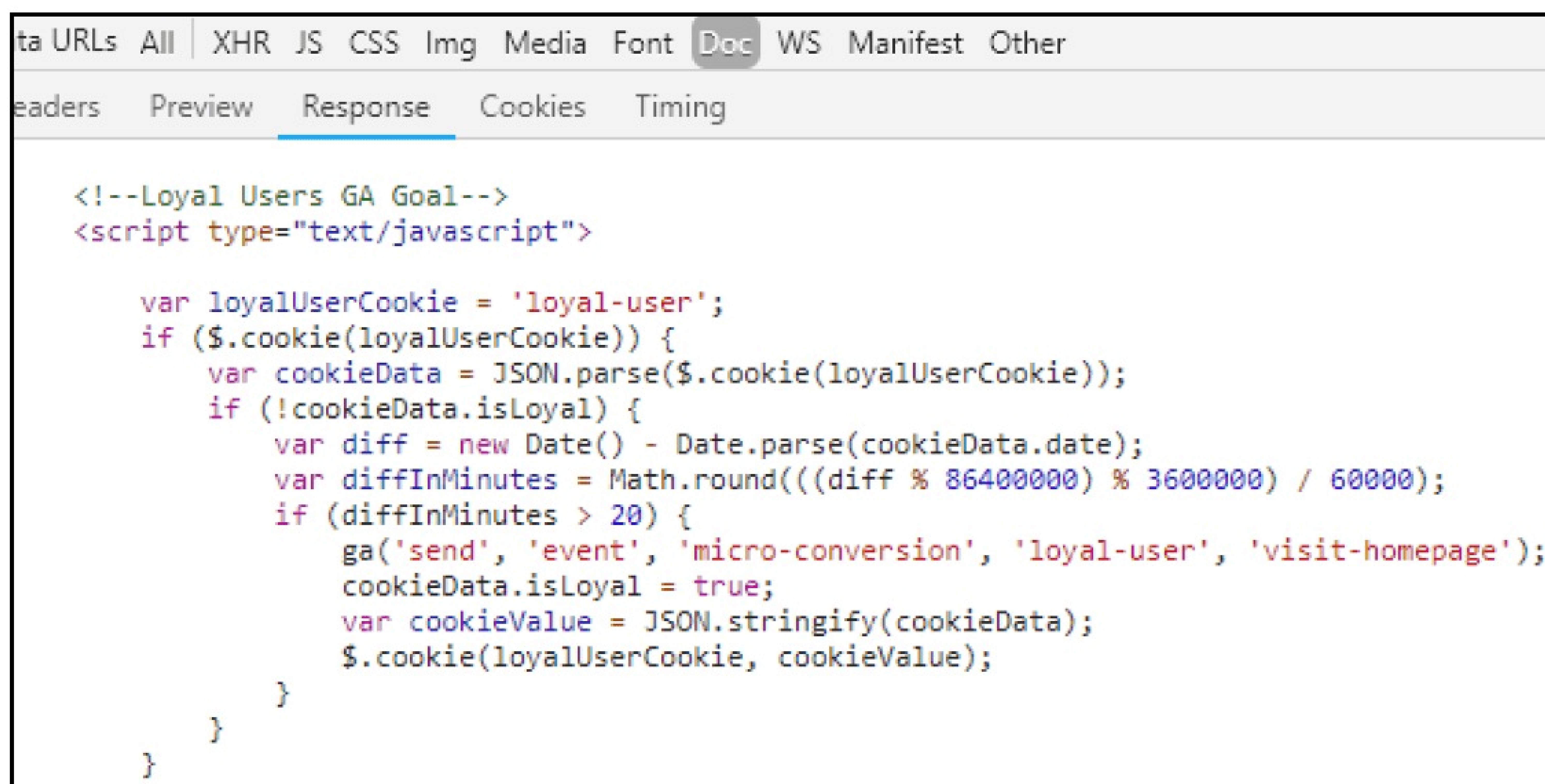


图 27-12 Cookies 的处理代码

4．利用浏览器获取 Cookies

解读 Cookies 的处理代码是非常耗时的，而且非常考验爬虫开发者对 JS 代码的熟练程度。此外，还可以利用浏览器获取 Cookies，使用 Requests-HTML、Selenium 或 Splash 等模块实现浏览器动态加载网页，然后从中获取 Cookies 信息。

第 18.2 节实现的 QQ 音乐歌曲下载是使用 Selenium 模拟浏览器加载网页，从而获取网页的 Cookies 信息。虽然这种方法可以轻松获取 Cookies 信息，但在浏览器加载网页的时候，它会自动加载网页的全部请求信息，因此也会降低爬虫的爬取速度。

27.6　本 章 小 结

反爬虫一直都是爬虫开发最让人头疼的问题。相信很多开发者会遇到这样的情况，在开发过程中，爬虫程序成功通过测试，但项目上线就会出现各种问题，比如 HTTP 请求出现异常或者无法从响应内容爬取目标数据等，这种“程序开发正常，上线出异常”的情况是因为网站设置了反爬虫机制。

（1）验证码是反爬虫机制里面最常用的手段之一，在爬虫中出现验证码的情况如下：

- 登录页面设有验证码识别。
- 特殊请求设置验证码。
- 网络环境导致验证码出现。

验证码识别的可行解决方案，说明如下：

- 使用 OCR 技术识别。
- 使用第三方平台 API 接口识别。
- 人为识别验证码。
- 使用 Selenium 控制程序实现识别验证码。
- 从浏览器复制 Cookies 并写入程序。

（2）爬虫是通过 HTTP 请求来获取目标数据，而请求参数是 HTTP 请求必不可少的构成部分，因此很多反爬虫机制都设置了请求参数，请求参数的数据来源方式如下：

- 参数值为固定可选值。
- 参数值来自其他请求的响应内容。
- 参数值经过 JS 处理。
- 参数值为特殊值。

根据请求参数的类型，可以总结出请求参数的查找方法，如下所示：

- 查找请求参数的变化规律。
- 从其他请求信息查找。
- 分析 JS 代码处理过程。
- 利用浏览器动态加载。

在请求头里设置反爬虫机制也是常见的手段之一，因此请求头的反爬虫机制如下：

- 检测请求头的固定属性。
- 检测请求头的可变属性。

（3）Cookies 是网站为了辨别用户身份、进行 Session 跟踪而储存在用户本地终端上的数据，一个 Cookies 就是存储在用户主机浏览器中的文本文件。网站利用 Cookies 设置反爬虫机制主要方式如下：

- 限制 Cookies 的访问频率（即限制用户的访问频率）。
- 限制 Cookies 的时效性。

在爬虫开发中，如果是 Cookies 引发的反爬虫机制，可以采取 4 种应对方法：

- 构建 Cookies 池。
- 使用代理 IP。
- 动态构建 Cookies。
- 利用浏览器获取 Cookies。

第 28 章

自己动手开发爬虫框架

28.1 框架设计说明

从本书的实战项目看到，爬虫开发不管是使用爬虫库还是爬虫框架，若按照功能划分，整个爬虫程序分为三部分：数据爬取、数据清洗和数据入库。本章开发的爬虫框架也是按照功能划分的逻辑来实现，目前尚处于雏形阶段，虽然能实现爬虫开发，但尚有很多功能有待完善。

本爬虫框架现由 4 个文件组成，分别是初始化文件__init__.py 和功能文件 pattern.py、spider.py、storage.py，文件说明如下：

- 初始化文件__init__.py 用于设置框架的版本信息和导入框架的功能文件。
- 数据清洗文件 pattern.py 用于定义数据清洗类，清洗方式与 Scrapy 框架相似。
- 数据爬取文件 spider.py 用于定义数据爬取类，爬取方式支持异步并发、URL 去重和分布式。
- 数据存储文件 storage.py 用于定义数据存储类，目前支持关系型数据库、非关系型数据库、CSV 文件存储数据和文件下载功能。

我们将框架命名为 pyReptile，在 D 盘里创建文件夹 pyReptile，然后在文件夹里创建文件，框架的目录结构如图 28-1 所示。

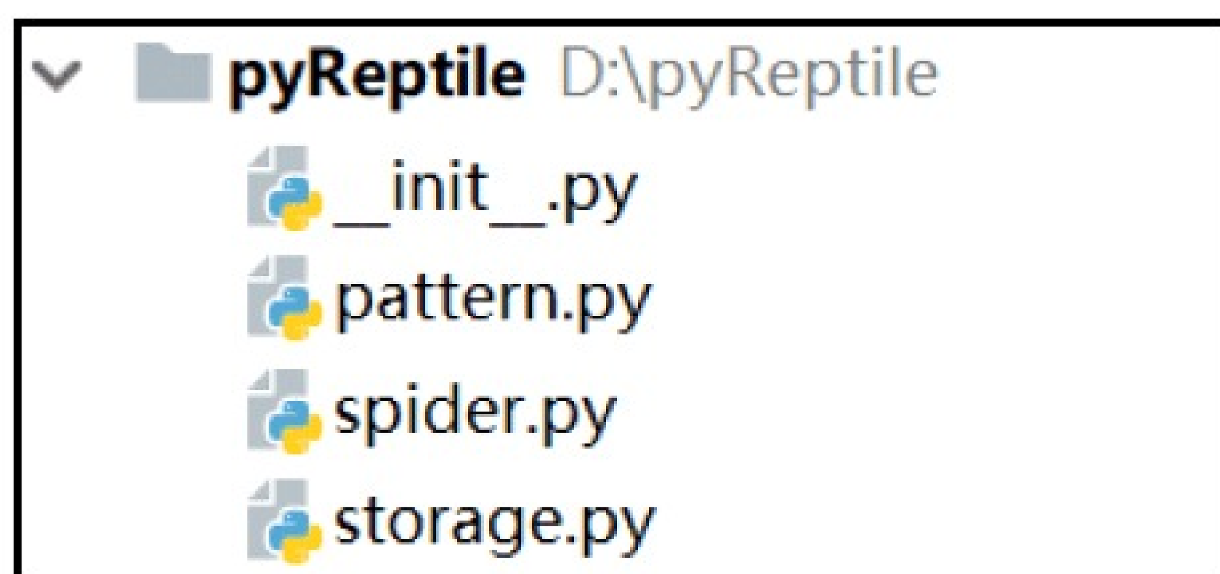


图 28-1 目录结构

由于初始化文件__init__.py 只是设置框架的版本信息及导入框架的功能文件，因此初始化文件

的代码如下：

```
# project: pyReptile
# author:  Xy Huang
  version   = '1.0.0'
# 导入功能文件
from .storage import *
from .spider import *
from .pattern import *
```

初始化文件是整个框架的入口，它导入了整个框架的功能。在使用框架的时候，只需在初始化文件调用相关的功能模块即可。功能文件 pattern.py、spider.py 和 storage.py 支撑整个框架的运行，其原理图如图 28-2 所示。

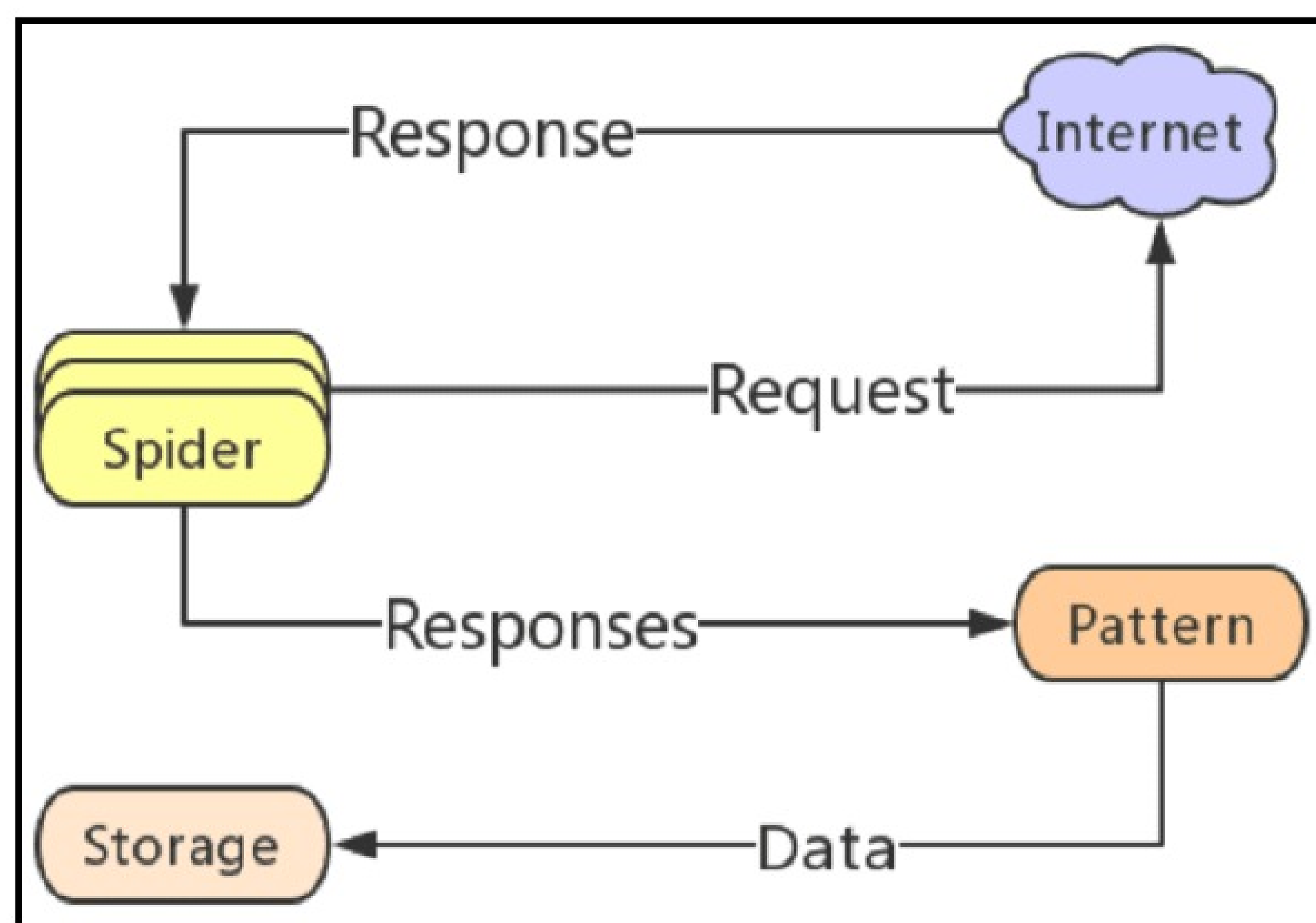


图 28-2　框架流程图

pyReptile 框架的设计原理是从 Scrapy 框架和 SQLAlchemy 框架受到启发的，具体的说明如下：

- 数据爬取方式由 URL 地址的数据格式决定，如果 URL 地址的数据格式为列表，pyReptile 就会执行异步并发，并将所有请求的响应内容以列表格式返回；如果传入的 URL 地址是字符串格式（即单一的 URL 地址），pyReptile 就直接返回相应的响应内容；并且还支持 URL 去重和分布式爬虫功能。
- 数据清洗采用 Scrapy 框架的清洗模式，使用方式与 Scrapy 框架有一定的相似之处，目前仅支持 CssSelector 和 Xpath 定位方式。
- 数据入库支持关系型数据库、非关系型数据库和 CSV 文件存储，关系型数据库由 SQLAlchemy 框架实现；非关系型数据库目前仅支持 MongoDB 数据库。pyReptile 简化入库方式，只需将爬取的数据以字典格式传入即可实现入库操作。

28.2　异步爬取方式

pyReptile 框架的数据爬取由 Aiohttp 模块实现，因此它具备了异步并发功能。我们将 Aiohttp 模块的数据爬取功能进行封装和延伸，简化了其使用方式，使用者只需调用相关的函数并传入参数

即可发送 HTTP 请求。打开 spider.py 文件，在文件里定义爬虫类 Request，代码如下：

```
import asyncio
import aiohttp
import redis

# 设置默认参数
TIMEOUT = 40
REQUEST_HEADERS = {
    'Accept': 'text/html,application/xhtml+xml,
             application/xml;q=0.9,*/*;q=0.8',
    'Accept-Language': 'en',
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                 AppleWebKit/537.36 (KHTML, like Gecko)
                 Chrome/69.0.3497.100 Safari/537.36'
}
# 实例化对象，用于发送 HTTP 请求
loop = asyncio.get_event_loop()

# 定义装饰器，实现 URL 去重或分布式处理
def distributes(func):
    def wrapper(self, url, **kwargs):
        redis_host = kwargs.get('redis_host', '')
        if redis_host:
            port = kwargs.get('port', 6379)
            db = kwargs.get('db', 1)
            redis_db = redis.Redis(host=redis_host, port=port, db=db)
            redis_data_dict = 'keys'
            if not redis_db.hexists(redis_data_dict, url):
                redis_db.hset(redis_data_dict, url, 0)
                return func(self, url, **kwargs)
            else:
                return {}
        else:
            return func(self, url, **kwargs)
    return wrapper

# 定义爬虫类
class Request(object):
    # 定义异步函数
    async def httpGet(self, url, **kwargs):
        cookies = kwargs.get('cookies', {})
        params = kwargs.get('params', {})
        proxy = kwargs.get('proxy', '')
        timeout = kwargs.get('timeout', TIMEOUT)
        headers = kwargs.get('headers', REQUEST_HEADERS)
        # 带代理 IP
        if proxy:
            async with aiohttp.ClientSession(cookies=cookies) as session:
                async with session.get(url, params=params, proxy=proxy,
                           timeout=timeout, headers=headers) as response:
                    # 将响应内容以字典格式返回
                    result = dict(
                        content=await response.read(),
```

```
                        text=await response.text(),
                        status=response.status,
                        headers=response.headers,
                        url=response.url
                    )
                    return result
        # 不带代理 IP
        else:
            async with aiohttp.ClientSession(cookies=cookies) as session:
                async with session.get(url, params=params, timeout=timeout,
                                       headers=headers) as response:
                    result = dict(
                        content=await response.read(),
                        text=await response.text(),
                        status=response.status,
                        headers=response.headers,
                        url=response.url
                    )
                    return result

    # 定义异步函数
    async def httpPost(self, url, **kwargs):
        cookies = kwargs.get('cookies', {})
        data = kwargs.get('data', {})
        proxy = kwargs.get('proxy', '')
        timeout = kwargs.get('timeout', TIMEOUT)
        headers = kwargs.get('headers', REQUEST HEADERS)
        if proxy:
            async with aiohttp.ClientSession(cookies=cookies) as session:
                async with session.post(url, data=data, proxy=proxy,
                               timeout=timeout, headers=headers) as response:
                    result = dict(
                        content=await response.read(),
                        text=await response.text(),
                        status=response.status,
                        headers=response.headers,
                        url=response.url
                    )
                    return result
        else:
            async with aiohttp.ClientSession(cookies=cookies) as session:
                async with session.post(url, data=data, timeout=timeout,
                                        headers=headers) as response:
                    result = dict(
                        content=await response.read(),
                        text=await response.text(),
                        status=response.status,
                        headers=response.headers,
                        url=response.url
                    )
                    return result

    # 定义 GET 请求方式
    @distributes
    def get(self, url, **kwargs):
```

```
        tasks = []
        if isinstance(url, list):
            for u in url:
                task = asyncio.ensure_future(self.httpGet(u, **kwargs))
                tasks.append(task)
            result = loop.run_until_complete(asyncio.gather(*tasks))
        else:
            result = loop.run_until_complete(self.httpGet(url, **kwargs))
        return result

    # 定义 POST 请求方式
    @distributes
    def post(self, url, **kwargs):
        tasks = []
        if isinstance(url, list):
            for u in url:
                task = asyncio.ensure_future(self.httpPost(u, **kwargs))
                tasks.append(task)
            result = loop.run_until_complete(asyncio.gather(*tasks))
        else:
            result = loop.run_until_complete(self.httpPost(url, **kwargs))
        return result

# 实例化 Request 对象
request = Request()
```

上述代码主要分为：初始化变量、定义装饰器与对象以及定义爬虫类 Request。初始化变量与对象是设置爬虫的超时时间、请求头以及实例化对象 loop，该对象用于发送 HTTP 请求；定义装饰器用于爬虫类 Request，实现 URL 去重功能或分布式功能。爬虫类 Request 一共定义 4 个函数，函数的功能说明如下：

- 函数 httpGet 是定义 Aiohttp 的异步 GET 请求函数，函数参数 url 以字符串格式表示，代表请求地址 URL，可选参数 kwargs 代表自定义的请求设置，如请求头、代理 IP、Cookies 信息、超时和请求参数等。
- 函数 httpGet 会对参数 proxy 进行判断，如果参数 proxy 非空，Aiohttp 在发送 GET 请求的时候，则在请求里添加参数 proxy，由于参数 proxy 的特殊性，如果参数 proxy 为空并且在请求里添加参数 proxy，Aiohttp 会提示异常信息，因此函数需要对参数 proxy 进行判断处理。最后，函数会将响应内容以字典格式返回。
- 函数 httpPost 是定义 Aiohttp 的异步 POST 请求函数，函数参数 url 和 kwargs 与函数 httpGet 的参数功能一致；函数的功能实现过程与函数 httpGet 的相似，区别在于两者的 HTTP 请求方式各有不同。
- 函数 get 是定义爬虫类 Request 的 GET 请求方式，函数参数 url 的数据格式可为字符串或列表格式，可选参数 kwargs 代表自定义的请求设置，如请求头、代理 IP、Cookies 信息、超时和请求参数等，参数 kwargs 也是函数 httpGet 的参数 kwargs。
- 函数 get 经过装饰器 distributes 过滤，装饰器从函数 get 获取 Redis 数据库连接参数，如果没有数据库连接参数，则往下执行函数 get；如果存在数据库连接参数，则连接 Redis 数据库并判断参数 url 是否记录在 Redis 数据库，若已记录，不再执行函数 get，反之执行函数 get。

- 函数 get 对参数 url 进行判断，如果 url 是列表，则对列表进行遍历，每次遍历调用函数 httpGet，传入当前的 URL 地址并添加到任务列表，然后将任务列表交给对象 loop 处理，对所有任务发送异步并发的 HTTP 请求，最后将所有请求的响应内容以列表格式返回。如果 url 是字符串，则由对象 loop 调用函数 httpGet，发送 HTTP 请求并返回响应内容。
- 函数 post 是定义爬虫类 Request 的 POST 请求方式，函数参数 url 和 kwargs 与函数 get 的参数功能一致；函数的功能实现过程与函数 get 的相似，区别在于两者调用的 Aiohttp 异步函数各有不同。

从爬虫类 Request 的代码可以看到，函数之间的代码存在重复使用的情况，因为 Aiohttp 在使用过程中需要以 with 模块化表示，从而导致代码出现重复。

为了测试爬虫类 Request 的功能是否正确，我们在 spider.py 文件目录下创建 spiderTest.py 文件，并在文件里编写功能测试代码，如下所示：

```
from spider import request
# GET 请求
from spider import request

# GET 请求
url = 'http://httpbin.org/get'
# url = ['http://httpbin.org/get']
params = {
    'pyReptile': 'spiderGet'
}
cookies = {
    'pyReptile': 'spiderCookies'
}
# URL 去重或分布式，设置 Redis 数据库连接参数
redis host = '127.0.0.1'

r = request.get(url, params=params, cookies=cookies,
                redis host=redis host)
print(r.get('text', ''))
# print(r[0]['text'])

# POST 请求
url = 'http://httpbin.org/post'
# url = ['http://httpbin.org/post']
data = {
    'pyReptile': 'spiderPost'
}
cookies = {
    'pyReptile': 'spiderCookies'
}
r = request.post(url, data=data, cookies=cookies)
print(r.get('text', ''))
# print(r[0]['text'])
```

上述代码简单演示了 pyReptile 框架的 GET 和 POST 请求，使用方法与 Requests 模块相似，但在发送 HTTP 请求的时候，pyReptile 框架会根据参数 url 的数据格式而执行相应的请求处理，这一优势是 Requests 模块无法比拟的。运行上述代码就会分别输出 GET 和 POST 请求的响应内容，如

图 28-3 所示。

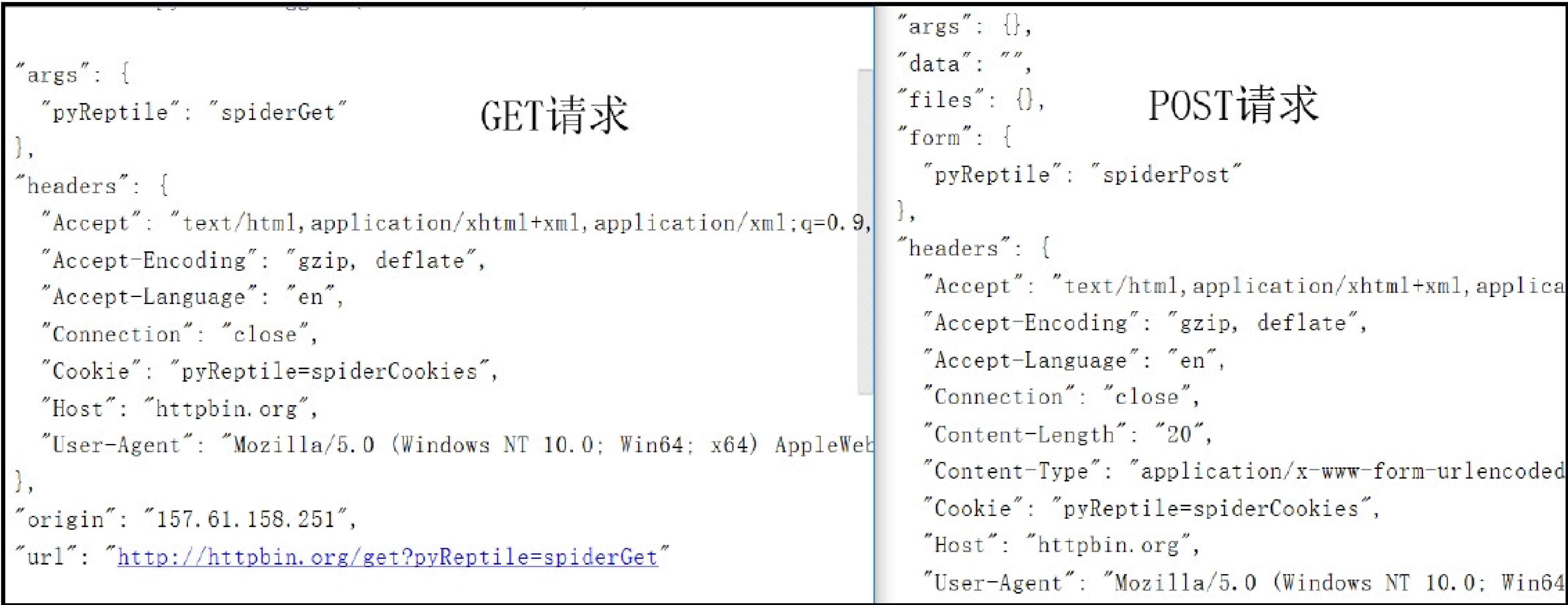


图 28-3　GET 和 POST 请求的响应内容

上述测试代码中，GET 请求设置数据库连接参数 redis_host，当再次运行上述代码时就不再执行 GET 请求，打开 RedisDesktopManager 查看 Redis 数据库，查看数据库所记录的 URL 地址，如图 28-4 所示。



图 28-4　Redis 数据库

28.3　数据清洗机制

pyReptile 框架的数据清洗由 BeautifulSoup4 和 lxml 模块实现，使用者只需调用相关的函数并传入相应的参数即可清洗数据。打开 pattern.py 文件，在文件里定义数据清洗类 DataPattern，代码如下：

```
from bs4 import BeautifulSoup
import lxml
from lxml.html.soupparser import fromstring as soup parse

class DataPattern(object):
    def cssSelector(self, response, selector, **kwargs):
        parser = kwargs.get('parser', 'html.parser')
        templist = []
```

```
        soup = BeautifulSoup(response, parser)
        temp = soup.select(selector=selector)
        for i in temp:
            tempList.append(i.getText())
        return tempList

    def xpath(self,response,selector, **kwargs):
        parser = kwargs.get('parser', 'html.parser')
        try:
            soup = soup parse(response, features=parser)
        except:
            soup = lxml.html.fromstring(response)
        temp = soup.xpath(selector)
        tempList = []
        for i in temp:
            tempList.append(i.text)
        return tempList

dataPattern = DataPattern()
```

数据清洗类 DataPattern 定义了函数 cssSelector()和 xpath()，两个函数的参数说明如下：

- 参数 response 代表 HTTP 请求的响应内容。
- 参数 selector 代表目标数据的定位方法，定位方法采用 5CssSelector 或 Xpath 语法。
- 可选参数 kwargs 是自定义设置，如参数 parser 可自定义选择 HTML 解析器，若无对参数 parser 进行设置，则默认使用 Python 标准库的 HTML 解析器——html.parser。

函数 cssSelector()和 xpath()实现数据清洗处理，具体的实现过程如下：

- 从可选参数 kwargs 获取参数 parser，如果 parser 的参数值为空，则默认使用 html.parser 作为解析器，将参数 response 的参数值进行 HTML 解析并生成 soup 对象。
- 由参数 selector 对 soup 对象进行定位和查找，从中找出符合条件的数据对象 temp。
- 遍历循环对象 temp，获取对象 temp 的数据内容并写入列表 tempList，再将列表作为函数返回值。
- 将数据清洗类 DataPattern 进行实例化，生成对象 dataPattern，用于开发者的调用。

为了测试数据清洗类 DataPattern 的功能是否正确，在 pattern.py 文件目录下创建 patternTest.py 文件，并在文件里编写功能测试代码，如下所示：

```
from pattern import dataPattern
from spider import request
url = 'https://movie.douban.com/subject/3168101/comments'
headers = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64)
                   AppleWebKit/537.36 (KHTML, like Gecko)
                   Chrome/70.0.3538.67 Safari/537.36'
}
r = request.get(url, headers=headers)

# cssSelector
title = dataPattern.cssSelector(r['text'], '#content > h1')
print(title)
```

```
    selector = 'div.comment> p > span'
    comment=dataPattern.cssSelector(r['text'],selector,parser='html5lib')
    print(len(comment))

    # xpath
    title = dataPattern.xpath(r['text'], '//*[@id="content"]/h1')
    print(title)
    selector = '//*[@id="comments"]//p//span'
    comment = dataPattern.xpath(r['text'], selector, parser='html5lib')
    print(len(comment))
```

上述代码使用爬虫类 Request 向豆瓣电影评论页发送 HTTP 请求，并将响应内容交给数据清洗对象 dataPattern 进行清洗处理，从响应内容中分别提取电影标题和评论内容。由于评论内容较多，我们只输出电影标题和评论总数，如图 28-5 所示。

```
['毒液：致命守护者 短评']
20
['毒液：致命守护者 短评']
20
```

图 28-5　数据清洗

注　意

函数 cssSelector()和 xpath()的参数 selector 必须遵从 CssSelector 和 Xpath 语法规则，有关 CssSelector 或 Xpath 语法规则，读者可以自行查阅资料。

28.4　数据存储机制

pyReptile 框架的数据存储是采用 SQLAlchemy 框架、pymongo 和 csv 模块实现的，分别提供了三种不同的数据存储方式，在使用过程中只需设置数据存储方式及调用相关方法即可实现数据存储处理。打开 storage.py 文件，在文件里定义数据存储类 DataStorage，代码如下：

```
    from sqlalchemy import *
    from sqlalchemy.orm import sessionmaker
    from sqlalchemy.ext.declarative import declarative base
    from pymongo import MongoClient
    import csv
    import os
    Base = declarative base()

    # 定义数据存储类 DataStorage
    class DataStorage(object):
        def   init  (self, CONNECTION, **kwargs):
            self.databaseType = kwargs.get('databaseType', 'CSV')
            # 根据参数 databaseType 选择存储方式，默认 CSV 文件存储
            if self.databaseType == 'SQL':
                # 根据字段创建映射类和数据表
```

```
            self.field()
            tablename = kwargs.get('tablename', self.  class  .  name  )
            self.table = self.table(tablename)
            self.DBSession = self.connect(CONNECTION)
        elif self.databaseType == 'NoSQL':
            self.DBSession = self.connect(CONNECTION)
        else:
            self.path = CONNECTION

    # 定义数据表字段
    def field(self):
        # self.name = Column(String(50))
        pass

    # 连接数据库，生成 DBSession 对象
    def connect(self, CONNECTION):
        # 连接关系型数据库
        if self.databaseType == 'SQL':
            engine = create engine(CONNECTION)
            DBSession = sessionmaker(bind=engine)()
            Base.metadata.create all(engine)
        # 连接非关系型数据库
        else:
            info = CONNECTION.split('/')
            # 连接 Mongo 数据库
            connection = MongoClient(
                info[0],
                int(info[1])
            )
            db = connection[info[2]]
            DBSession = db[info[3]]
        return DBSession

    # 定义映射类
    def table(self, tablename):
        class TempTable(Base):
              tablename   = tablename
            id = Column(Integer, primary key=True)
        # 将类属些进行判断，符合 sqlalchemy 的字段则定义到数据映射类
        for k, v in self.  dict  .items():
            if isinstance(v, Column):
                setattr(TempTable, k, v)
        return TempTable

    # 插入数据
    def insert(self, value):
        # 关系型数据库的数据插入
        if self.databaseType == 'SQL':
            self.DBSession.execute(self.table.  table  .insert(), value)
            self.DBSession.commit()
        # 非关系型数据库的数据插入
        elif self.databaseType == 'NoSQL':
            # 判断参数 value 的数据类型，选择单条数据还是多条数据插入
            if isinstance(value, list):
```

```
                self.DBSession.insert many(value)
            else:
                self.DBSession.insert(value)

    # 更新数据
    def update(self, value, condition={}):
        # 关系型数据库的数据更新
        if self.databaseType == 'SQL':
            # 更新条件只设置了单个条件
            if condition:
                c = self.table.  dict  [list(condition.keys())[0]].
                    in (list(condition.values()))
                self.DBSession.execute(self.table.  table  .
                    update().where(c).values(), value)
            # 全表更新
            else:
                self.DBSession.execute(self.table.__table__.
                    update().values(), value)
            self.DBSession.commit()
        # 非关系型数据库的数据更新
        elif self.databaseType == 'NoSQL':
            self.DBSession.update many(condition, {'$set': value})

    # 文件下载
    def getfile(self, content, filepath):
        with open(filepath, 'wb') as code:
            code.write(content)

    # 数据写入 CSV 文件
    def writeCSV(self, value, title=[]):
        # 参数 title 为空列表，则将字典的 keys 进行排序并作为 CSV 的标题
        if not title:
            title = sorted(value[0].keys())
        # 判断文件是否存在，
        pathExists = os.path.exists(self.path)
        with open(self.path, 'a', newline='') as csv file:
            csv writer = csv.writer(csv file)
            # 文件不存在，则写入标题
            if not pathExists:
                csv writer.writerow(title)
            # 将数据写入 CSV 文件
            for v in value:
                valueList = []
                for t in title:
                    valueList.append(v[t])
                csv_writer.writerow(valueList)
```

数据存储类 DataStorage 定义 8 个方法，分别是初始化方法__init__()、类方法 field()、connect()、table()、insert()、update()、getfile()和 writeCSV()，每个方法所实现的功能说明如下：

（1）初始化方法__init__()根据参数 databaseType 来执行相应的数据存储方式，每种数据存储方式说明如下：

- 如果参数 databaseType 设为 SQL，则说明数据存储方式为关系型数据库。初始化方法会从可选参数 kwargs 里获取参数 tablename，如果参数 tablename 不存在，则由子类的名字作为数据表的表名；然后调用类方法 field()，从类方法 field()里获取自定义的字段属性，用于定义数据表映射类；再调用类方法 table()来创建数据表映射类，并以类属性 table 表示；最后调用类方法 connect()进行数据库连接，将数据库连接对象返回并以类属性 DBSession 表示。
- 如果参数 databaseType 设为 NoSQL，则说明数据存储方式为非关系型数据库。初始化方法只调用类方法 connect()并把参数 CONNECTION 传入，实现数据库连接，将数据库连接对象返回并以类属性 DBSession 表示。
- 如果参数 databaseType 设为 CSV 或没有设置参数 databaseType，则说明数据存储方式为 CSV 文件存储。初始化方法将参数 CONNECTION 赋值给类属性 path，类属性 path 代表 CSV 文件路径信息。

（2）类方法 field()让开发者自定义数据表字段，主要用于关系型数据库的存储方式。在使用过程中，通过子类继承数据存储类 DataStorage，在子类里重写类方法 field()即可实现自定义表字段。

（3）类方法 connect()根据参数 databaseType 来选择相应的数据库连接方式。如果使用关系型数据库，则使用 SQLAlchemy 框架实现数据库连接，反之则使用 pymongo 模块连接 MongoDB。

（4）类方法 table()定义数据表映射类 TempTable，映射类会默认创建主键 ID，然后遍历数据存储类 DataStorage 的类属性，并对每个类属性的数据类型进行判断，如果类属性是 Column 对象（即 SQLAlchemy 的表字段对象），则使用 Python 内置方法 setattr()将类属些写入数据表映射类 TempTable。

（5）类方法 insert()实现数据入库功能，支持关系型和非关系型数据库的数据入库操作。插入的数据必须是字典格式，并且字典的 key 必须为表字段。参数 value 可以是列表或字典形式，若是以字典表示，则插入单条数据，若是以列表表示，则插入多条数据。

（6）类方法 update()实现数据更新功能，支持关系型和非关系型数据库的数据更新操作。参数 value 必须是字典格式，并且字典的 key 必须为表字段；参数 condition 是更新条件，它的默认值为 None，如果参数值为 None，则对全表数据进行更新处理，反之对符合条件的数据进行更新处理。

（7）类方法 getfile()实现文件下载功能，参数 content 代表文件内容；参数 filepath 代表文件所保存的绝对路径。

（8）类方法 writeCSV()实现 CSV 文件存储数据功能，参数 title 代表文件表头内容，如果参数值为空，则以参数 value 首个元素的 keys 作为表头内容，参数 title 以列表表示，列表元素决定了数据写入顺序；参数 value 是待存储的数据内容，也是以列表表示，每个列表元素是以字典表示。

综上，类方法 field()、connect()和 table()主要用于初始化方法__init__()，为初始化方法__init__()分别提供数据表字段、数据库连接对象 DBSession 和数据表映射类 TempTable；类方法 insert()和 update()是实现数据库的数据操作（如数据的新增或修改）；getfile()和 writeCSV()分别实现文件下载功能和 CSV 文件存储数据功能。

为了验证数据存储类 DataStorage 的功能是否正确，在 storage.py 文件目录下创建三个测试文件 storageTest-CSV.py、storageTest-NoSQL.py 和 storageTest-SQL.py，分别验证三种数据存储方式。

首先打开 storageTest-CSV.py，在文件里编写功能测试代码，验证 CSV 文件存储数据功能，如下所示：

```
from storage import *

if   name   == '  main  ':
    CONNECTION = 'data.csv'
    # 待存储数据 personInfo
    personInfo = [{'name': 'Lucy', 'age': '21', 'address': '北京市'},
                  {'name': 'Lily', 'age': '18', 'address': '上海市'}]
    # 实例化数据存储类 DataStorage
    database = DataStorage(CONNECTION)
    # 调用 writeCSV()实现 CSV 文件存储
    # database.writeCSV(personInfo)
    database.writeCSV(personInfo, title=['name', 'age', 'address'])
```

变量 CONNECTION 是 CSV 文件路径信息，在实例化数据存储类 DataStorage 的时候传入变量 CONNECTION 即可将数据存储方式选为 CSV 文件存储，无须设置参数 databaseType。实例化对象 database 调用 writeCSV()方法即可实现 CSV 文件存储数据功能。

运行上述代码，并控制参数 title 的传入方式，分别查看参数 title 的传入是否对文件存储的造成影响，如图 28-6 所示。

name	age	address
Lucy	21	北京市
Lily	18	上海市
带参数title		

	address	age	name
1	address	age	name
2	北京市	21	Lucy
3	上海市	18	Lily
4			
5	不带参数title		

图 28-6　CSV 文件存储

接着打开 storageTest-NoSQL.py，在文件里编写功能测试代码，验证非关系型数据库的数据存储功能，如下所示：

```
from storage import *

if   name   == '  main  ':
    CONNECTION = 'localhost/27017/test/storage db'
    # 实例化数据存储类 DataStorage
    database = DataStorage(CONNECTION, databaseType='NoSQL')
    # 插入多条数据
    personInfo = [{'name': 'Lucy', 'age': '21', 'address': '北京市'},
                  {'name': 'Lily', 'age': '18', 'address': '上海市'}]
    database.insert(personInfo)
    # 插入单条数据
    value = {'name': 'Tom', 'age': '21', 'address': '北京市'}
    database.insert(value)
    # 更新数据
    condition = {'name': 'Lucy'}
    updateInfo = {'name': 'Lucy', 'age': '22', 'address': '广州市'}
    database.update(updateInfo, condition)
```

变量 CONNECTION 是 MongoDB 的连接方式，在实例化数据存储类 DataStorage 的时候，传入变量 CONNECTION 并设置参数 databaseType 为 NoSQL 即可选择非关系型数据库的数据存储功能。实例化对象 database 调用 insert()和 update()方法，分别实现多条数据插入、单条数据插入和数据更新功能。

运行上述代码之前，在 MongoDB 的可视化工具里操作 MongoDB，创建数据库 test。代码运行成功后，在可视化工具里查看数据库 test 的 storage_db 集合，该集合的数据信息如图 28-7 所示。

MyDb　localhost:27017　test

db.getCollection('storage_db').find({})

storage_db　0.001 sec.

	_id	name	age	address
1	ObjectId("5c21d9117...	Lucy	22	广州市
2	ObjectId("5c21d9117...	Lily	18	上海市
3	ObjectId("5c21d9117...	Tom	21	北京市

图 28-7　非关系型数据库的数据存储功能

最后打开 storageTest-SQL.py，在文件里编写功能测试代码，验证关系型数据库的数据存储功能，如下所示：

```
from storage import *

# 定义数据表 personinfo
class PersonInfo(DataStorage):
    def field(self):
        # 定义数据表字段
        # self.name = Column(String(50))
        self.name = Column(String(50), comment='姓名')
        self.age = Column(String(50), comment='年龄')
        self.address = Column(String(50), comment='地址')

# 定义数据表 schoolinfo
class SchoolInfo(DataStorage):
    def field(self):
        # 定义数据表字段
        # self.name = Column(String(50))
        self.school = Column(String(50), comment='学校')
        self.name = Column(String(50), comment='姓名')

if __name__ =='__main__':
    CONNECTION = 'mysql+pymysql://root:1234@
                  localhost/storage_db?charset=utf8mb4'
    person = PersonInfo(CONNECTION, databaseType='SQL')
    school = SchoolInfo(CONNECTION, databaseType='SQL')
    # 对 personInfo 表插入多条数据
    personInfo = [{'name': 'Lucy', 'age': '21', 'address': '北京市'},
```

```
                {'name': 'Lily', 'age': '18', 'address': '上海市'}]
    person.insert(personInfo)
    # 对 schoolInfo 表插入单条数据
    schoolInfo = {'name': 'Lucy', 'school': '清华大学'}
    school.insert(schoolInfo)

    # 对 personInfo 表更新数据
    condition = {'id': 1}
    personInfo = {'name': 'Lucy', 'age': '22', 'address': '广州市'}
    person.update(personInfo, condition)
    # 对 schoolInfo 表更新数据
    schoolInfo = {'name': 'Lucy', 'school': '北京大学'}
    school.update(schoolInfo, condition)
```

上述代码分别定义了数据存储类 PersonInfo 和 SchoolInfo，两者通过重写类方法 field()来实现表字段的定义。在文件中的运行函数__main__分别对类 PersonInfo 和 SchoolInfo 进行实例化，由于子类继承了父类 DataStorage 的初始化方法，因此数据存储类 PersonInfo 和 SchoolInfo 在实例化的时候会定义数据表映射类和创建数据表连接对象，最后实例化对象 person 和 school 分别调用 insert()和 update()方法，实现数据的入库和更新处理。

从使用方式发现，关系型数据库的使用方式不同于非关系型数据库和 CSV 文件，前者是通过定义子类并继承数据存储类 DataStorage，再实例化子类并调用相关的方法，从而实现数据存储功能；而非关系型数据库和 CSV 文件是直接实例化数据存储类 DataStorage 并调用相关的方法。

运行上述代码，并打开数据库 storage_db 查看数据表 schoolinfo 和 personinfo 的数据信息，如图 28-8 所示。

id	school	name
1	北京大学	Lucy

id	name	age	address
1	Lucy	22	广州市
2	Lily	18	上海市

图 28-8　数据表 schoolinfo 和 personinfo 的数据信息

28.5　实战：用自制框架爬取豆瓣电影

相信读者对 pyReptile 框架设计已有一定的了解，本节我们通过一个实战项目来讲述如何使用 pyReptile 框架实现爬虫开发。以豆瓣电影为例，选取某一部电影作为爬取对象，分别爬取电影信息和电影评论。在电影信息页（https://movie.douban.com/subject/3168101/?from=showing）里分别爬取电影名称和剧情简介，如图 28-9 所示。

图 28-9　电影信息页

然后在浏览器中打开电影评论页（https://movie.douban.com/subject/3168101/comments?status=P），分别爬取用户名和评论内容，如图 28-10 所示。

图 28-10　电影评论页

爬取的数据皆可从开发者工具 Network 选项卡的 Doc 分类标签里找到数据位置，本节不再讲述网页结构的分析过程。我们将 pyReptile 框架放置在 Python 安装目录的 site-packages 文件夹，这是将 pyReptile 框架以第三方库的形式安装在 Python 里，如图 28-11 所示。

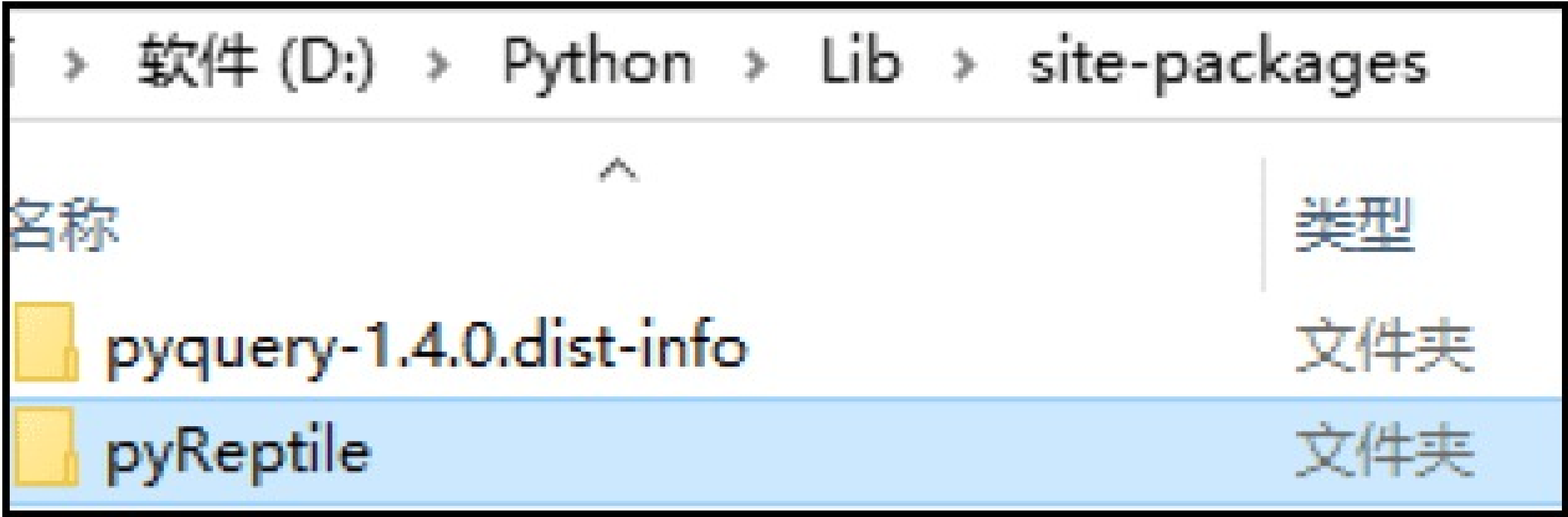


图 28-11　安装 pyReptile 框架

完成pyReptile框架安装后，在D盘下创建文件夹doubanSpider，并在文件夹里分别创建fields.py和 spider.py 文件。文件夹 doubanSpider 是项目的文件目录，如图 28-12 所示。



图 28-12　项目文件目录

打开 fields.py 文件，分别定义数据存储类 MovieComment 和 MovieInfo，两者皆继承 pyReptile 框架的数据存储类 DataStorage。在自定义的数据存储类中，重写类方法 field()并在类方法里自定义类属性，每个自定义的类属性代表数据表的表字段，代码如下：

```
from pyReptile.storage import *
# 定义电影信息表的字段
class MovieComment(DataStorage):
    def field(self):
        # 定义数据表字段
        self.movieId = Column(String(50), comment='电影 ID')
        self.user = Column(String(50), comment='用户名')
        self.comment = Column(String(3000), comment='评论内容')

# 定义电影评论表的字段
class MovieInfo(DataStorage):
    def field(self):
        # 定义数据表字段
        self.movieId = Column(String(50), comment='电影 ID')
        self.name = Column(String(50), comment='电影名称')
        self.summary = Column(String(3000), comment='剧情简介')
```

最后在 spider.py 文件里编写具体的爬虫规则，数据存储介质选择 MySQL 数据库，爬取数据是某部电影的基本信息和前十页的评论内容，实现代码如下：

```
from pyReptile import request, dataPattern
from fields import MovieComment, MovieInfo
import time

# 基本设置
CONNECTION = 'mysql+pymysql://root:1234@localhost/
```

```
                spiderdb?charset=utf8mb4'
# 实例化数据存储类，定义映射类以及创建数据表
movieComment = MovieComment(CONNECTION)
movieInfo = MovieInfo(CONNECTION)

# 爬取电影信息
def get_movie(movieId):
    # URL 以字符串格式传入
    r = request.get(movieUrl % (movieId))
    name = dataPattern.cssSelector(r['text'], 'h1 > span')[0]
    summary = dataPattern.cssSelector(r['text'],'#link-report')[0].strip()
    movieDic = dict(movieId=movieId, name=name, summary=summary)
    # 查询数据表是否已存在数据
    queryMovie = movieInfo.DBSession.query(movieInfo.table).
                 filter_by(movieId=movieId).all()
    # 存在数据则作更新处理
    if queryMovie:
        condition = {'movieId': movieId}
        movieInfo.update(movieDic, condition)
    # 不存在就插入新的数据
    else:
        movieInfo.insert(movieDic)

# 爬取电影评论
def get_comment(movieId):
    # URL 以列表格式传入
    urlList = []
    for page in range(10):
        urlList.append(commentUrl % (movieId, str(page * 20)))
    valueList = []
    responseList = request.get(urlList)
    for response in responseList:
        commentList = dataPattern.cssSelector(response['text'],
                      'div.comment > p > span')
        userList = dataPattern.cssSelector(response['text'],
                   'span.comment-info > a')
        for comment, user in zip(commentList, userList):
            valueList.append(dict(movieId=movieId, user=user,
                             comment=comment))
    # 数据入库
    movieComment.insert(valueList)
if __name__ == '__main__':
    # 开始时间
    localTime = time.localtime(time.time())
    beginTime = time.strftime("%H:%M:%S", localTime)
    print('程序开始时间：' + beginTime)
    # 爬虫程序
    movieUrl = 'https://movie.douban.com/subject/%s/?from=showing'
    commentUrl = 'https://movie.douban.com/subject/%s/comments?
                 start=%s&limit=20&sort=new_score&status=P'
    movieId = '3168101'
    get_movie(movieId)
    get_comment(movieId)
    # 结束时间
```

```
    localTime = time.localtime(time.time())
    endTime = time.strftime("%H:%M:%S", localTime)
    print('程序结束时间：' + endTime)
```

上述代码可划分为 4 部分，分别是 pyReptile 框架功能的初始化、电影信息的爬虫函数 get_movie()、电影评论的爬虫函数 get_comment()和文件运行入口，说明如下：

（1）pyReptile 框架功能的初始化是设置 SQLAlchemy 连接 MySQL 的连接内容，由 pymysql 模块实现连接，数据存储在数据库 spiderdb；将数据库的连接内容以参数的形式传入数据存储类 MovieComment 和 MovieInfo，生成实例化对象 movieComment 和 movieInfo。

（2）电影信息的爬虫函数 get_movie()是对电影信息页进行数据爬取、清洗和入库处理，说明如下：

- 首先对电影信息页的 URL 地址发送 HTTP 请求，因为只爬取某一部电影，所以 URL 地址是以字符串格式表示。
- 从响应内容里提取电影名称和剧情简介，将提取的数据转换成字典格式，字典的 key 是数据表的表字段，即数据存储类 MovieInfo 定义的类属些，字典的 value 是提取的数据内容。
- 最后由对象 movieInfo 判断电影 ID 是否已存在，若存在，则对数据表的数据进行更新处理，反之则对数据表新增数据。

（3）电影评论的爬虫函数 get_comment()是对前十页的电影评论页进行数据爬取、清洗和入库处理，说明如下：

- 前十页的电影评论页共有 10 条不同的 URL 地址，因此 URL 地址是以列表的形式传入请求函数 get()，pyReptile 框架对其执行异步并发的 HTTP 请求。
- 将前十页的响应内容进行遍历，每次遍历会提取当前页面的用户名和评论内容，再将用户名和评论内容转换成字典格式，并且写入列表 valueList，该列表保存了前十页所有的用户名和评论内容。
- 最后由对象 movieComment 对列表 valueList 执行数据入库处理。

（4）文件运行入口是设置电影 ID、信息页和评论页的 URL 地址、调用爬虫函数 get_movie()和 get_comment()以及设置程序运行的开始时间和结束时间。通过程序运行前后的时间对比，可以得知 pyReptile 框架的爬取效率。运行 spider.py 文件，若不考虑网速或硬件等因素，项目的爬取效率约为 3 秒，如图 28-13 所示。

```
程序开始时间：17:36:33
程序结束时间：17:36:36
```

图 28-13　爬取效率

最后打开数据库 spiderdb，分别查看数据表 movieinfo 和 moviecomment 的数据信息，如图 28-14 所示。

图 28-14　数据表 movieinfo 和 moviecomment 的数据信息

28.6　本 章 小 结

爬虫框架现由 4 个文件组成，分别是初始化文件__init__.py、功能文件 pattern.py、spider.py 和 storage.py，文件说明如下：

- 初始化文件__init__.py 是设置框架的版本信息及导入框架的功能文件。
- 数据清洗文件 pattern.py 是定义数据清洗类，清洗方式与 Scrapy 框架相似。
- 数据爬取文件 spider.py 是定义数据爬取类，爬取方式支持异步并发、URL 去重和分布式。
- 数据存储文件 storage.py 是定义数据存储类，目前支持关系型数据库、非关系型数据库、CSV 文件存储数据和文件下载功能。

spider.py 实现初始化变量、定义装饰器与对象和定义爬虫类 Request。初始化变量与对象是设置爬虫的超时时间、请求头以及实例化对象 loop，该对象用于发送 HTTP 请求；定义装饰器是用于爬虫类 Request，实现 URL 去重功能或分布式功能。爬虫类 Request 定义 4 个函数：httpGet()、httpPost()、get()和 post()。

pattern.py 定义数据清洗类 DataPattern，它是由 BeautifulSoup4 和 lxml 模块实现的，使用者只需调用相关的函数并传入相应的参数即可清洗数据。

storage.py 定义数据存储类 DataStorage，采用 SQLAlchemy 框架、pymongo 和 csv 模块实现，提供三种不同的数据存储方式，在使用过程中只需设置数据存储方式以及调用相关方法即可实现数据存储处理。

爬虫框架目前还有很多功能尚未完善，比如爬虫类 Request 需要添加 Selenium 或 Splash 等功能、数据清洗类 DataPattern 和数据存储类 DataStorage 的运行逻辑尚不成熟。若读者有兴趣参与 pyReptile 框架的开发，可以联系笔者。